

Тема: Remote Method Invocation (Part 3)

План занятия:

1. [Удаленные ссылки на несколько удаленных интерфейсов](#)
2. [Завершение работы RMI приложения](#)
3. [Динамическая активация объектов](#)
 - [Основная идея](#)
 - [Реализация](#)
 - [Протокол активации](#)
 - [Активатор](#)
 - [Группа активации](#)
 - [Простой демонстрационный пример](#)
 - [Удаленный интерфейс](#)
 - [Класс, реализующий удаленный интерфейс](#)
 - [Создание класса сервера](#)
 - [Создание класса клиента](#)
 - [Разверывание, компиляция и запуск приложения](#)
 - [Второй демонстрационный пример](#)
 - [Удаленный интерфейс](#)
 - [Класс, реализующий удаленный интерфейс](#)
 - [Активация удаленного объекта](#)
 - [Клиентская часть приложения](#)
 - [Разверывание, компиляция и запуск 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. Java™ Remote Method Invocation Specification. 7 Remote Object Activation:
<https://docs.oracle.com/javase/9/docs/specs/rmi/activation.html>

9. rmid - The Java RMI Activation System Daemon:

<https://docs.oracle.com/javase/7/docs/technotes/tools/solaris/rmid.html>

10. Jan Graba *An Introduction to Network Programming with Java*, 2013

11. Роберт Орфали, Дан Харки. *Java и CORBA в приложениях клиент-сервер*. Издательство: Лори, 2000 - 734 с.

12. Патрик Нимейер, Дэниэл Леук. *Программирование на Java* (Исчерпывающее руководство для профессионалов). М.: Эксмо, 2014

13. William Grosso. *Java RMI*. O'Reily, 2001, p. 752

14. Esmond Pitt, Kathleen McNiff. *java(TM).rmi: The Remote Method Invocation Guide*. Addison-Wesley, 2001, p. 320

Удаленные ссылки на несколько удаленных интерфейсов

Класс удаленного объекта может реализовать несколько удаленных интерфейсов. При передаче удаленной ссылки на такой удаленный объект другой виртуальной машине получатель получает заглушку, имеющую доступ к удаленным методам, объявленным в каждом из удаленных интерфейсов. Для того чтобы выяснить, реализует ли такой интерфейс конкретный удаленный объект, можно воспользоваться рефлексией, например операцией `instanceof`. Если проверка проходит успешно, переменную удаленной ссылки можно привести к проверенному типу и вызвать соответствующий метод.

Напишем приложение таким образом, чтобы его можно было запускать из среды разработки. При этом теряется впечатление, что мы работаем на разных компьютерах, но, при желании, можно запустить это приложение и обычным для *RMI* приложений образом. Но, вообще-то в нашем примере сосредоточимся на особенностях применения удаленных объектов, класс которых имплементирует несколько удаленных интерфейсов.

Рассмотрим небольшой пример, показывающий особенности работы с удаленным объектом, класс которого имплементирует несколько удаленных интерфейсов.

Определим два удаленных интерфейса с одним методом каждый. Удаленный интерфейс `Hello`.

```
package multiple;

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

public interface Hello extends Remote {
    public String sayHello(String name) throws
        RemoteException;
}
```

Удаленный объект будет принадлежать классу, который имплементирует этот интерфейс, а также интерфейс `GoodBye`:

```
package multiple;

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

public interface GoodBye extends Remote {
    public String sayGoodBye() throws RemoteException;
}
```

Создадим класс реализации, имплементирующий два указанных интерфейса:

```
package multiple;

import java.rmi.RemoteException;

public class HelloGoodbyeImpl implements Hello, GoodBye {

    private int numberOfVisitors;

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

    @Override
    public String sayGoodBye() throws RemoteException {
        return "Good Bye. The interaction is finished (" +
            this.numberOfVisitors + " visitors).";
    }

    @Override
    public String sayHello(String name) throws
        RemoteException {
        return "Hello, " + name + "! Your number is " +
            (++this.numberOfVisitors);
    }
}
```

Также, в соответствии с рассмотренной ранее схемой, создадим серверную часть приложения:

```
package multiple;

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

```

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

public class ServerRMI {

    private HelloGoodbyeImpl greeter = null;

    public ServerRMI() {
        this.greeter = new HelloGoodbyeImpl();
    }

    public void start() {
        export_register(1099);
    }

    public void start(int port) {
        export_register(port);
    }

    private void export_register(int port) {
        try {
            Registry registry =
                LocateRegistry.createRegistry(port);
            Remote stub =
                UnicastRemoteObject.exportObject(this.greeter, 0);
            registry.rebind("greeter", stub);
            System.out.println(
                "The greeter object is ready to work (use port " + port +
                    ")");
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        ServerRMI rmi = new ServerRMI();
        rmi.start(6789);
    }
}

```

Следует обратить внимание на то, что в данном серверном приложении служба *RMI* реестра программно создается при старте серверной части и с завершает свою работу при завершении работы сервера.

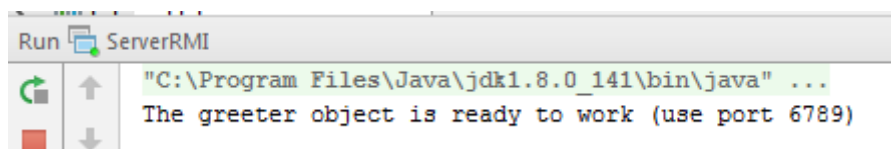
Когда удаленная ссылка на такой удаленный объект передается на другую виртуальную машину, то у получателя есть заглушка, которая имеет

доступ к удаленным методам, объявленным в интерфейсах Hello и GoodBye. Для безопасности, можно применить оператор instanceof, чтобы узнать, реализует ли конкретный удаленный объект запрашиваемый интерфейс. Например, предположим, что получен удаленный объект через переменную типа GoodBye.

GoodBye greet = register....

Удаленный объект, на который указывает ссылка stub, может или не может быть приведен к типу Hello. Чтобы это узнать, используйте тест if (stub instanceof Hello). Если тест пройден, вы можете привести приветствие к типу Hello и вызвать метод sayHello.

Из среды разработки запустим серверную часть приложения:



Рассмотрим несколько вариантов клиентов для нашего распределенного приложения.

Клиент первого типа:

```
package multiple;

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

public class HelloClient {
    public static void main(String[] args) {
        String host = "localhost";
        int port = 6789;
        String url = "rmi://" + host + ":" + port;
        try {
            String[] list = Naming.list(url);
            System.out.println("Remote objects: " +
                               Arrays.toString(list));
            String name = "greeter";
            Hello stub = (Hello) Naming.lookup(url+"/"+name);
            System.out.println(stub.sayHello("Vasisyalij"));
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (MalformedURLException e) {
```

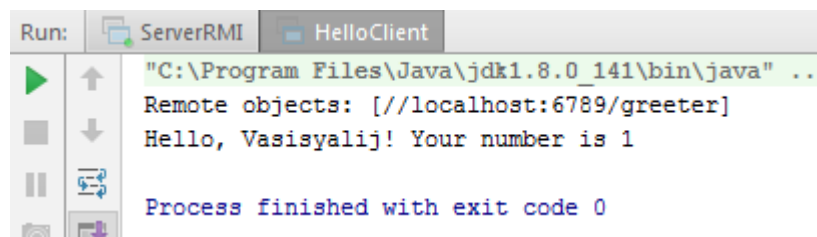
```

        e.printStackTrace();
    } catch (NotBoundException e) {
        e.printStackTrace();
    }
}
}

```

Клиент данного типа построен по стандартной схеме. Следует обратить внимание на то, что вначале серверное приложение с помощью вызова метода `Naming.list(url)` получает список всех зарегистрированных удаленных объектов в службе реестра в виде *URL*-подобной строки без указания схемы и выводит его на экран. В данном варианте удаленного клиента на удаленном объекте можно вызвать методы одного интерфейса (в данном случае `Hello`). Если класс удаленного объекта не будет имплементировать этот интерфейс, по будет выброшено исключение и программа завершит свою работу.

Запустим эту клиентскую часть распределенного приложения:



Программа выполнит ожидаемые действия.

Рассмотрим код клиента, реализованного немного по другой схеме:

```

package multiple;

import java.rmi.NotBoundException;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class GoodByeClient {
    public static void main(String[] args) {
        String host = "localhost";
        int port = 6789;
        String name = "greeter";
        try {
            Registry registry =
LocateRegistry.getRegistry(host, port);
            Remote stub = registry.lookup(name);
            if (stub instanceof GoodBye) {
                System.out.println("Ok. It is right type");
                GoodBye obj = (GoodBye) stub;

```

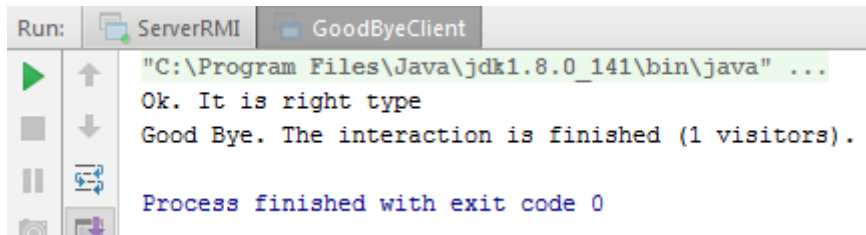
```

        System.out.println(obj.sayGoodBye());
    } else {
        System.out.println("The type is wrong!");
    }
} catch (RemoteException e) {
    e.printStackTrace();
} catch (NotBoundException e) {
    e.printStackTrace();
}
}
}

```

В данном варианте клиента удаленная ссылка поступает в приложение от службы реестра в виде объекта типа `Remote`. Затем выполняется проверка – а имплементирует ли удаленный объект требуемый удаленный интерфейс – если да, то выполняется сужающее преобразование типов и вызывается требуемый метод удаленного интерфейса.

Запустим этот вариант клиентской части распределенного приложения:



Рассмотрим еще один вариант кода клиентской части.

В данном варианте клиента распределенного приложения удаленная ссылка поступает в приложение от службы реестра также в виде объекта типа `Remote`, а вызов метода выполняется с помощью рефлексии (см. сборник лабораторных работ: Лабораторная работа №2. Задание №3). Выполняется попытка найти метод с заданным именем. Если она оказывается неудачной, то выбрасывается исключение собственного типа, а если метод найден – то он вызывается средствами рефлексии. Обратите внимание на вспомогательные классы, реализующие такое поведение.

```

package multiple;

import multiple.reflect.Invoker;
import multiple.reflect.MethodNotFoundException;
import multiple.reflect.RDArray;
import multiple.reflect.ReflectPair;

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

```

```

import java.rmi.registry LocateRegistry;
import java.rmi.registry Registry;

public class ClientRMI {
    public static void main(String[] args) {
        String host = "localhost";
        int port = 6789;
        String name = "greeter";
        try {
            Registry registry =
                LocateRegistry.getRegistry(host, port);
            Remote stub = registry.lookup(name);
            RDArray arr = new RDArray();
            arr.add(new ReflectPair(String.class, "Berlaga"));
            try {
                Object res = Invoker.invoke(stub,
                                                "sayHello", arr);
                System.out.println(res);
                res = Invoker.invoke(stub, "sayGoodBye");
                System.out.println(res);
            } catch (MethodNotFoundException e) {
                e.printStackTrace();
            }
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (NotBoundException e) {
            e.printStackTrace();
        }
    }
}

```

Рассмотрим вспомогательные классы из занятия по рефлексии:

```

package multiple.reflect;

import java.lang.reflect.AccessibleObject;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.util.Arrays;

public class Invoker {

    public static Object invoke(Object obj, String name)
        throws MethodNotFoundException {

```



```

Object res = null;

try {
    Method m = obj.getClass().getDeclaredMethod(name,
                                                    null);
    if (!Modifier.isPublic(m.getModifiers())) {
        m.setAccessible(true);
    }
    try {
        res = m.invoke(obj, null);
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
} catch (NoSuchMethodException e) {
    throw new MethodNotFoundException(e,
                                       name + " Not Found");
} catch (SecurityException e) {
    e.printStackTrace();
}
return res;
}

public static Object invoke(Object obj, String name,
                             RArray arr) throws MethodNotFoundException {
    Object res = null;
    int len = arr.size();
    Class<?>[] cls = new Class[len];
    Object[] data = new Object[len];
    for (int i = 0; i < len; i++) {
        cls[i] = arr.get(i).getCls();
        data[i] = arr.get(i).getData();
    }

    try {
        Method m = obj.getClass().getDeclaredMethod(name,
                                                       cls);
        if (!Modifier.isPublic(m.getModifiers())) {
            m.setAccessible(true);
        }
        try {
            res = m.invoke(obj, data);
        } catch (IllegalAccessException e) {

```

```

        e.printStackTrace();
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
} catch (NoSuchMethodException e) {
    throw new MethodNotFoundException(e,
                                      name + " Not Found");
} catch (SecurityException e) {
    e.printStackTrace();
}
return res;
}

public static void main(String[] args) throws
                                MethodNotFoundException {
    TestClass tc = new TestClass();
    System.out.println(tc);
    RDArray arr = new RDArray();
    arr.add(new ReflectPair(double.class, 1));
    System.out.println(Invoker.invoke(tc, "score", arr));
    RDArray arr1 = new RDArray();
    arr1.add(new ReflectPair(double.class, 1));
    arr1.add(new ReflectPair(int.class, 1));
    System.out.println(Invoker.invoke(tc, "score", arr1));
}
}

```

```
package multiple.reflect;
```

```

public class MethodNotFoundException extends Exception {

    private static final long serialVersionUID = 1L;

    private NoSuchMethodException e;
    private String str;

    public MethodNotFoundException(NoSuchMethodException e,
                                   String str) {

        super(str);
        this.e = e;
        this.str = str;
    }
}

```

```
}

    public NoSuchMethodException getException() {
        return e;
    }

    public String getMessage() {
        return str;
    }
}

}

package multiple.reflect;

public class ReflectPair {

    private Class<?> cls;
    private Object data;

    public ReflectPair(Class<?> cls, Object data) {
        super();
        this.cls = cls;
        this.data = data;
    }
    public Class<?> getCls() {
        return cls;
    }
    public Object getData() {
        return data;
    }
}

}
```

```
package multiple.reflect;

import java.util.ArrayList;
import java.util.List;

public class RDArrary {

    private List<ReflectPair> lst = null;

    public RDArrary() {
```

```

    lst = new ArrayList<ReflectPair>();
}

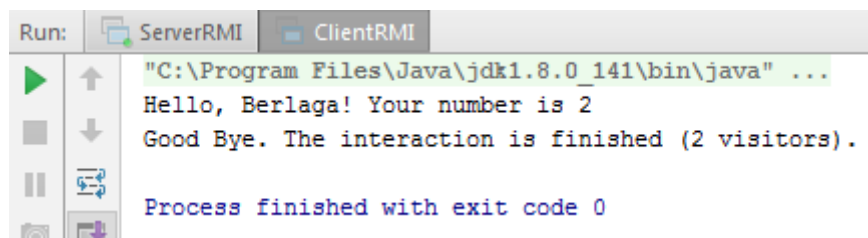
public void add(ReflectPair p) {
    lst.add(p);
}

public ReflectPair get(int ind) {
    return lst.get(ind);
}

public int size() {
    return lst.size();
}
}

```

Запустим этот вариант клиентской части распределенного приложения:



Приложение отработает ожидаемым образом. После всего, завершим средствами интегрированной среды разработки работу серверной части приложения.

Завершение работы RMI приложения

Все наши серверные классы постороены по одному принципу. Создается удаленный объект, выполняется его экспорт и регистрация заглушки в службе *RMI* реестра. Как только серверная часть приложения регистрирует удаленный объект в локальном *RMI* реестре, метод `main` завершает свою работу. Хотя метод `main()` и завершается сразу же после регистрации удаленного объекта, но приложение все еще остается в рабочем состоянии. Дело в том, что создании объекта класса, расширяющего класс `UnicastRemoteObject` (либо когда выполняется явная операция экспорта с помощью статического метода этого класса) запускается отдельный поток исполнения, удерживающий серверную программу в рабочем состоянии. Благодаря этому удаленный объект, созданный на сервере, и далее остается доступным, позволяя клиентам выполнять на нем вызовы удаленных методов через систему *RMI*. Кстати, из-за такого неявного использования многопоточности рекомендуется делать удаленные объекты потоковобезопасными (*thread safe*).

Для нормального завершения работы серверной части приложения нужно выполнить действия, по сути, обратные тем, которые были сделаны для регистрации удаленного объекта. Нужно отметить регистрацию удаленных объектов в службе *RMI* реестра при помощи вызова метода `unbind()` на объекте интерфейса `Registry` (см. <https://javadoc.scijava.org/Java8/java/rmi/registry/Registry.html>). В качестве параметра методу передается строковое имя объекта в реестре для удаления. После того, как выполнена операция «отвязывания» всех удаленных объектов из службы реестра, необходимо выполнить операцию де-экспорта удаленного объекта с помощью статического метода класса `UnicastRemoteObject.unexportObject(obj, true)` (см. <https://docs.oracle.com/javase/8/docs/api/java/rmi/server/UnicastRemoteObject.html>). Данный метод удаляет удаленный объект `obj` из среды выполнения *RMI*. В качестве первого параметра нужно указывать именно удаленный объект, а не его заглушку. В случае успешного вызова объект `obj` больше не будет принимать входящие вызовы *RMI*. Если для второго параметра метода задано значение `true`, объект принудительно де-экспортируется, даже если на удаленном объекте выполняются удаленные вызовы или есть ожидающие вызовы; если второй параметр метода имеет значение `false`, объект де-экспортируется, если только нет ожидающих или выполняющихся на нем удаленных вызовов. В этом случае все потоки исполнения среды выполнения *RMI*, связанные с этим удаленным объектом, завершают свою работу. И серверная часть приложения завершает свою работу. Если реестр *RMI* был создан программно, то это также удаленный объект, который для завершения работы серверной части приложения должен быть де-экспортирован. Если же для запуска службы реестра было запущено внешнее (по отношению к программе) приложение `rmiregistry`, то операцию де-экспорта выполнять не нужно, а приложение `rmiregistry` следует завершить средствами операционной системы.

Приведем в качестве примера одну из возможных реализаций нашего демонстрационного «вежливого» распределенного приложения с измененным кодом сервера, который корректно завершит свою работу.

Приведем код примера целиком. Как обычно сначала определяем удаленный интерфейс.

```
package test_stop;

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

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

Потом создадим класс, имплементирующий этот удаленный интерфейс.

```

package test_stop;

public class Greeter implements Greeting {

    private int numberOfVisitors;

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

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

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

```

Приведем стандартный код клиента, который сначала получает список всех имен объектов, зарегистрированных в реестре (метод `list()`, вызванный на объекте типа `Registry`, см. <https://docs.oracle.com/javase/8/docs/api/java/rmi/registry/Registry.html>), а затем получает записку для первого в списке удаленного объекта и с ее помощью вызывает удаленный метод.

```

package test_stop;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class GreetClient {
    public static void main(String[] args) {
        int port = 1099;
        String host = "localhost";
        System.out.println("Host: " + host + "\tPort: " + port);
        Registry registry = null;
        try {
            registry = LocateRegistry.getRegistry(host, port);

```

```

        System.out.println("Stub Names:");
        String [] names = registry.list();
        for (String name : names) {
            System.out.println("\t"+name);
        }
        Greeting stub = (Greeting) registry.lookup(names[0]);
        String name = "Vasisyalij Lohankin";
        System.out.println(stub.greet(name));
    } catch (RemoteException | NotBoundException e) {
        e.printStackTrace();
    }
}
}

```

Наконец, создадим исходный код класса, создающего удаленный объект, программно создающего службе *RMI* реестра и регистрирующего удаленный объект в этой службе. Дополнительно к этому, в классе реализован метод, выполняющий обратные операции - операцию «отвязывания» удаленного объекта от *RMI* реестра, операцию «де-экспорта» удаленного объекта и операцию «де-экспорта» программно созданного *RMI* реестра (см. метод `stop()`).

```

package test_stop;

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

public class GreetServer {
    private int port;
    private Greeter obj = null;
    private Registry registry = null;

    public GreetServer(int port) throws RemoteException {
        this.port = port;
        this.obj = new Greeter();
        registry = LocateRegistry.createRegistry(port);
        System.out.println("The RMI registry is created...");
        Remote stub = UnicastRemoteObject.exportObject(obj,0);
        System.out.println("The object is exported...");
        registry.rebind(obj.getClass().getSimpleName(), obj);
    }
}

```

```

        System.out.println("The stub is registered in RMI registry...");
        System.out.println("The Greeter server is ready to work...");
    }

    public GreetServer() throws RemoteException {
        this(1099);
    }

    public void stop() throws RemoteException, NotBoundException {
        registry.unbind(obj.getClass().getSimpleName());
        System.out.println("The stub is unregistered from RMI registry...");
        if (UnicastRemoteObject.unexportObject(obj,true)) {
            System.out.println("The object is unexported...");
        } else {
            System.out.println("The object is Not unexported...");
        }
        if (UnicastRemoteObject.unexportObject(registry,true)) {
            System.out.println("The registry is unexported...");
        } else {
            System.out.println("The registry is Not unexported...");
        }
    }
}

```

Ну и собственно класс, в котором создается серверная часть распределенного приложения, выполняется ожидание в течение заданного интервала времени (10 сек), когда сервер ожидает запросов на вызов методов от клиентов и завершение работы сервера по истечении этого интервала времени.

```

package test_stop;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;

public class Server {
    public static void main(String[] args) {
        int sec = 10;
        try {
            GreetServer server = new GreetServer();
            System.out.println("\nThe server will work about " +
                               sec + " seconds\n");
        }
    }
}

```



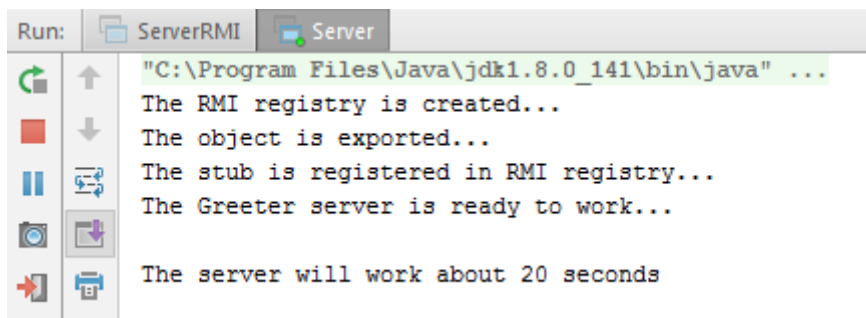
```

        Thread.sleep(sec*1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    server.stop();
} catch (RemoteException e) {
    e.printStackTrace();
} catch (NotBoundException e) {
    e.printStackTrace();
} finally {
    System.out.println("The server is stopped...");
}
}
}

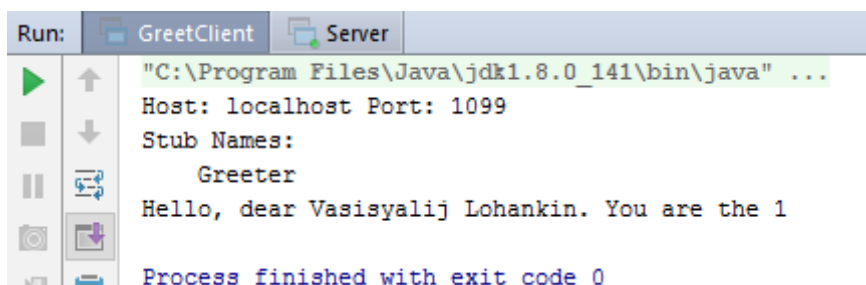
```

Запустим на выполнение это приложение непосредственно из интегрированной среды разработки (использовалась *IntelliJ Idea*). Конечно, по способу запуска это не совсем распределенное приложение (клиентская и серверная части в одном каталоге), но для нас важно посмотреть не запуск приложения (его мы отработали ранее), а особенности завершения работы серверной части.

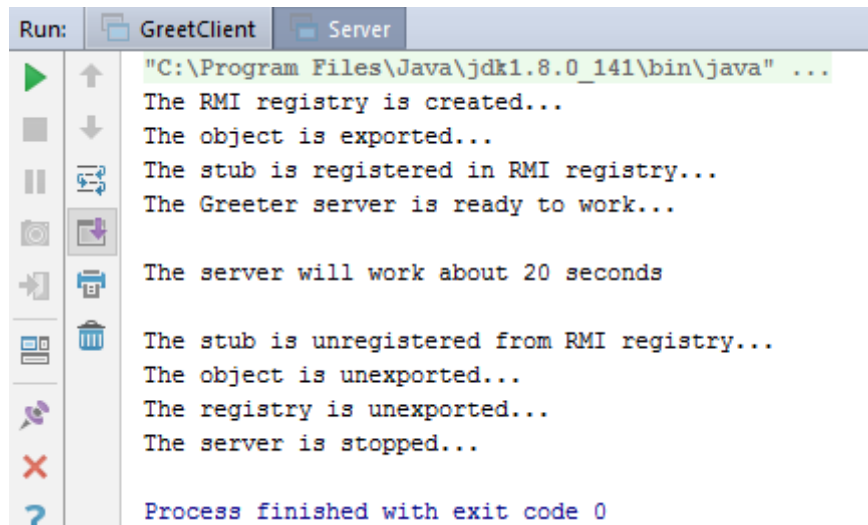
Запускаем серверную часть приложения. На экран выводятся информационные сообщения сервера.



При дано запуске время работы сервера было увеличено до 20 сек. В течение этого времени от ожидаемо обрабатывает запросы клиентов на вызов удаленного метода.



По истечении заданного интервала времени (20 сек.) вызывается метод, останавливающий сервер.



```
Run: GreetClient Server
"C:\Program Files\Java\jdk1.8.0_141\bin\java" ...
The RMI registry is created...
The object is exported...
The stub is registered in RMI registry...
The Greeter server is ready to work...
The server will work about 20 seconds
The stub is unregistered from RMI registry...
The object is unexported...
The registry is unexported...
The server is stopped...
Process finished with exit code 0
```

Удаленные объекты (у объект класса, имплементирующего удаленный интерфейс, и объект *RMI* реестра) удаляются из среды выполнения *RMI* и серверная часть приложения завершает работу.

Динамическая активация объектов

В тех примерах, которые мы до сих пор разбирали, удаленные объекты должны были быть созданы при старте приложения, и должны были существовать все время, пока это приложение работает, даже если они не нужны при данном запуске. Кроме такого способа работы с удаленными объектами, в системе времени выполнения *RMI* можно регистрировать информацию о реализациях удаленных объектов, экземпляр которых должен создаваться и выполняться по требованию, а не запускаться при старте приложения и существовать постоянно во время его работы. В данном случае система *RMI* не будет создавать экземпляр объекта до тех пор, пока не поступит запрос на вызов удаленного метода. Это значит, что в системе работает *ленивый (отложенный) активатор (lazy activator)*. См. <https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/activation/overview.html>.

Основная идея

При использовании *ленивой (отложенной) схемы активации*, серверная программа заранее не создает экземпляр удаленного объекта. Вместо этого приложению под названием *rmid* (демон активации, входит в стандартную поставку *JDK*, см.

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/rmid.html>)

предоставляется информация о реализации удаленного объекта (*remote object implementation*) (такая, как полное имя класса-файла этого объекта; месторасположение, из которого класс может быть загружен; данные, которые будут использоваться для начальной загрузки объекта (*object bootstrap*) и т. д.). В это время удаленный объект находится в неактивном *inactive* (или

пассивном *passive*) состоянии. Однако даже для такого пассивного состояния объекта (объект еще предстоит создать и экспортировать), его заглушка (*stub*) может быть получена из этого приложения *rmid* и привязывается *bound* к *RMI* реестру таким образом, чтобы клиенты могли получить удаленную ссылку из этого реестра. Когда приходит запрос на вызов удаленного метода, демон активации *rmid* проверяет существование базового объекта. Если объект еще не существует, то создается новый экземпляр на основании той информации, которая была предоставлена системе *RMI* во время процедуры регистрации. Теперь этот вновь созданный объект находится в активном *active* состоянии. Процесс трансформации пассивного объекта в активный объект называется *активацией activation*. Далее, этот активный объект используется для реализации вызова удаленного метода.

Реализация

См. <https://docs.oracle.com/javase/8/docs/platform/rmi/spec/rmi-activation.html#a1997>

Система *RMI* реализует *отложенную активацию* при помощи *faulting remote reference* (ошибочной удаленной ссылки), которую еще по-другому называют *fault block* (блоком ошибок). Так, на практике, заглушка удаленного объекта содержит *faulting remote reference* (ошибочную удаленную ссылку), которая состоит из:

- *activation identifier* (идентификатор активации) для удаленного объекта;
- *transient reference* (ссылка для прехода), ссылающаяся на активный (*active*) удаленный объект.

Идентификатор активации (*activation identifier*) содержит информацию, достаточную для активации объекта. Ссылка для перехода (*transient reference*) - это действительная активная ссылка на активный удаленный объект. Первоначально действующая ссылка на удаленный объект равна *null*, и это указывает на то, что объект еще не активен. Когда приходит первый запрос на вызов удаленного метода, тогда ошибочная ссылка проверяет действующую ссылку и обнаруживает, что объект находится в пассивном состоянии. В этом случае она инициирует *протокол активации* (*activation protocol*, см. чуть позже) для активации объекта с помощью идентификатора активации и устанавливает текущую ссылку на только что активированный удаленный объект. Как только *ошибочная ссылка* (*faulting reference*) получает *действующую ссылку* (*live reference*), она перенаправляет вызов удаленного метода на *базовую удаленную ссылку* (*underlying remote reference*), которая, в свою очередь, перенаправляет вызов метода на удаленный объект.

Протокол активации (*activation protocol*)

Протокол активации определяет то, каким образом следует активировать *активируемый пассивный объект* (*activatable passive object*). Протокол, при выполнении активации, использует такие компоненты: *ошибочная ссылка*

(*faulting reference*, см. прошлый раздел), *активатор* (*activator*), *группа активации* (*activation group*) и объект, который нужно активировать.

Активатор (*Activator*)

Этот компонент не активирует удаленный объект, а контролирует весь процесс активации. Основные задачи, которые должен решить активатор, заключаются в следующем:

- поддерживать базу данных с информацией, которая необходима для активации объекта (например, имя класс-файла объекта; местоположение, из которого класс может быть загружено; данные, необходимые для начальной загрузки объекта и т. д.);
- перенаправлять запросы на активацию объекта (совместно со всей необходимой информацией) в правильную *группу активации* (*activation group*) внутри удаленной *JVM*;
- при необходимости управлять запускаемыми виртуальными машинами *Java*.

Группа активации (*Activation group*)

Группа активации – это тот компонент, который фактически активирует удаленный объект в соответствии с информацией, указанной для активации, и возвращает обратно активатору активированный объект. Говорят, что объект принадлежит к указанной группе активации. Рассмотрим основные этапы процесса активации объекта.

Сначала *ошибочная ссылка* (*faulting reference*) вызывает активатор (интерфейс RMI см.

<https://docs.oracle.com/javase/8/docs/api/java/rmi/activation/Activator.html>)

передавая ему *идентификатор активации* (*activation identifier*), чтобы активировать тот объект, который связан с переданным идентификатором. *Активатор* (*activator*) обращается к *дескриптору активации объекта* (*object's activation descriptor*), хранящему следующую информацию:

- *идентификатор группы* (*group identifier*) объекта, который определяет ту *JVM*, где должен быть активирован объект;
- полное имя класса активируемого удаленного объекта;
- местоположение в формате *URL*, откуда можно загрузить определение класса для активируемого объекта;
- данные, необходимые для инициализации объекта в *маршаллизованной* (*упакованной, marshalled*) форме.

Активатор направляет запрос на активацию группе активации (если она уже существует), к которой должен относиться этот объект. Если группа активации еще не создана, то активатор создает новую группу активации и затем перенаправляет запрос на активацию в эту группу. Группа активации считывает информацию активации из дескриптора активации объекта, загружает класс для объекта и создает экземпляр объекта, используя специальный конструктор с двумя аргументами.

После того, как требуемый объект был активирован, группа активации передает *марshallизованную* (*marshalled*) ссылку на объект обратно в активатор. Затем активатор записывает пару действующая ссылка и идентификатор активации и возвращает *действующую* (*активную, live, active*) ссылку на *ошибочную ссылку* (*faulting reference*). Ошибочная ссылка (*faulting reference*) на заглушку, наконец, перенаправляет вызов удаленного метода через *действующую ссылку* (*live reference*) непосредственно на удаленный объект.

Простой демонстрационный пример

Рассмотрим простой вариант процедуры динамической активации объектов на простом примере распределенного калькулятора, выполняющего четыре простых арифметических действия. По смыслу этот пример очень похож на наше «вежливое» приложение, только проще в смысле активации. Рассмотрим основные моменты по созданию приложения, обратив особое внимание на:

- измененный класс реализации удаленного интерфейса;
- измененный класс сервера;
- измененную процедуру запуска приложения.

Удаленный интерфейс

В нашем приложении удаленный объект предоставляет четыре метода, реализующих основные алгебраические операции, каждый из которых принимает два целых числа в качестве аргументов и возвращает целое число в качестве результата.

Интерфейс, который соответствует такому описанию, может иметь такой вид:

```
package activation;

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

public interface Calculator extends Remote {
    public int add(int a, int b) throws RemoteException;
    public int subtract(int a, int b) throws RemoteException;
    public int multiply(int a, int b) throws RemoteException;
    public int divide(int a, int b) throws RemoteException;
}
```

Класс, реализующий удаленный интерфейс

Обычно, основным встроенным классом *Java* при реализации технологии динамической активации объектов *RMI* является класс `java.rmi.activation.Activatable` (<https://docs.oracle.com/javase/8/docs/api/java/rmi/activation/Activatable.html>), собственно и обеспечивающий поддержку активации удаленных объектов.

Класс реализации активируемого удаленного объекта обычно расширяет этот класс (а если нет, то можно как и раньше выполнить явный экспорт объекта с помощью статического метода `exportObject()`). Теперь заголовок класса реализации будет таким:

```
package activation;

public class RemoteCalculator extends Activatable
    implements Calculator {

}
```

Данный класс, должен содержать конструктор с двумя параметрами (идентификатор активации и объект класса `MarshaledObject`, который содержит информацию для корректной инициализации полей объекта), который нужен для корректной активации объекта.

```
public RemoteCalculator(ActivationID id,
    MarshalledObject data) throws
    RemoteException {

    super(id, 0);
    System.out.println("SimpleCalculator Instantiated.");
}
```

Первым действием явно указывается вызов конструктора базового класса, в котором передача значения 0 в качестве второго параметра означает, что RMI-библиотека должна сама найти и присвоить подходящий номер порта для приема обращений. В данном примере из-за того, что по-сути объект класса, реализующего удаленный интерфейс, не имеет полей, то объект класса `MarshaledObject` явно не используется.

Кроме того, в классе должны быть реализованы все методы удаленного интерфейса. Рассмотрим возможный исходный код класса:

```
package activation;

import java.rmi.MarshaledObject;
import java.rmi.RemoteException;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;

public class RemoteCalculator extends Activatable
    implements Calculator {

    public RemoteCalculator(ActivationID id,
        MarshalledObject data) throws
        RemoteException {
```

```

        super(id, 0);
        System.out.println("SimpleCalculator Instantiated.");
    }

    @Override
    public int add(int a, int b) {
        System.out.println("Operation add. Received: " +
                           a + " and " + b);
        int result = a + b;
        System.out.println("Sent: " + result);
        return result;
    }

    @Override
    public int subtract(int a, int b) {
        System.out.println("Operation subtract. Received: " +
                           a + " and " + b);
        int result = a - b;
        System.out.println("Sent: " + result);
        return result;
    }

    @Override
    public int multiply(int a, int b) {
        System.out.println("Operation multiply. Received: " +
                           a + " and " + b);
        int result = a * b;
        System.out.println("Sent: " + result);
        return result;
    }

    @Override
    public int divide(int a, int b) {
        System.out.println("Operation divide. Received: " +
                           a + " and " + b);
        int result = a / b;
        System.out.println("Sent: " + result);
        return result;
    }
}

```

Создание класса сервера

Сам сервер нашего распределенного приложения не создает экземпляры класса `RemoteCalculator`. Вместо этого он собирает необходимую информацию, которая нужна для создания активируемого удаленного объекта. Затем сервер пересылает эту информацию в запущенный заранее демон

активации `rmid`, получает ссылку на экземпляр класса-заглушки активируемого класса и, наконец, регистрирует эту заглушку в `rmiregistry`. После этого работа сервера заканчивается. Впоследствии клиент может обратиться к `rmiregistry` для того, чтобы получить удаленную ссылку.

В рамках рассматриваемой технологии предполагается, что каждый активируемый объект должен принадлежат к какой-то *группе активации* (*activation group*). Именно группа активации отвечает за активацию объекта по требованию. Для этого нужно получить *идентификатор группы активации*. Группа активации может уже существовать или быть новой, создаваемой по запросу (если требуемая не существует). Поэтому нужно создать *дескриптор группы активации* (*activation group descriptor*). Назначение дескриптора группы активации состоит в том, чтобы предоставить демону активации `rmid` всю информацию, необходимую или для установления связи с соответствующей существующей *JVM*, или для создания новой *JVM* для активируемого объекта. Он включает в себя следующую информацию:

- имя класса создаваемой группы;
- месторасположение, откуда может быть загружено определение класса группы;
- «маршализированный» объект, содержащий данные для начальной загрузки для указанной группы.

Дескриптор группы активации создается с использованием класса `ActivationGroupDesc` (см. <https://docs.oracle.com/javase/8/docs/api/java/rmi/activation/ActivationGroupDesc.html>). Он содержит два конструктора:

```
ActivationGroupDesc(Properties overrides,
                    ActivationGroupDesc.CommandEnvironment cmd)
ActivationGroupDesc(String className, String location,
                    MarshalledObject<?> data, Properties overrides,
                    ActivationGroupDesc.CommandEnvironment cmd)
```

Первый конструктор (который и будет использован в нашем приложении) создает дескриптор группы активации, используя реализацию группы по умолчанию и месторасположение кода по умолчанию. Параметр `Properties` используются для переопределения системных свойств в виртуальной машине при реализации группы. Параметр `CommandEnvironment` управляет форматом команды и параметрами, используемыми для запуска дочерней виртуальной машины. Вместо этого можно указать значение `null`, если будут использованы значения по умолчанию программы `rmid`. Дескриптор группы активации можно создать так:


```
Properties props = new Properties();
String path = new File("policy").getCanonicalPath();
props.put("java.security.policy", path);
ActivationGroupDesc aGroup = new
    ActivationGroupDesc(props, null);
```

В данном фрагменте создается дескриптор группы активации, при этом используются системные значения по умолчанию для имени класса реализации группы, месторасположения кода и особенностей командной среды. Поскольку программа `rmid` работает в изолированной «песочнице» (*security sandbox*), то нужно определить политику безопасности, которая будет храниться в файле «`policy`». Для простоты воспользуемся файлом политики, который никому ничего не запрещает (в реальных приложениях именно такой файл не используется):

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

Этот дескриптор группы активации затем нужно передать демону активации `rmid`, который должен быть уже запущен и работать на некотором порту (порт по умолчанию для службы активации 1098, см. <https://docs.oracle.com/javase/7/docs/technotes/tools/solaris/rmid.html>). Это можно сделать таким образом:

```
ActivationGroupID id =
    ActivationGroup.getSystem().registerGroup(aGroup);
```

Метод `getSystem()` возвращает ссылку на `ActivationSystem` (см. <https://docs.oracle.com/javase/8/docs/api/java/rmi/activation/ActivationSystem.html>, <https://docs.oracle.com/javase/8/docs/api/java/rmi/activation/ActivationGroup.html>, <https://docs.oracle.com/javase/8/docs/api/java/rmi/activation/ActivationGroupID.html>), которая затем регистрирует указанный дескриптор группы активации. Затем приложение `rmid` или связывается с соответствующей существующей *JVM*, или создает новую *JVM* для активируемого объекта и возвращает идентификатор `id`, представляющий группу. Этот идентификатор затем используется для создания дескриптора активации активируемого объекта следующим образом:

```
String location = "http://localhost:8090/RMI/";
ActivationDesc desc = new ActivationDesc(id,
    RemoteCalculator.class.getName(),
    location, null);
```

Задача дескриптора активации (см. <https://docs.oracle.com/javase/8/docs/api/java/rmi/activation/ActivationDesc.html>) - предоставить всю информацию, которая необходима программе `rmid` для создания нового экземпляра класса, реализующего удаленный интерфейс. Вторым аргументом - это имя класса реализации (в нашем случае получаем его с помощью механизма рефлексии) для активируемого объекта. Если предположить, что месторасположение класса `SimpleCalculator` совпадает с каталогом начального запуска программы `rmid`, третий аргумент метода можно указать как `null`. Если же месторасположение определения класса отличается, от каталога начального запуска демона активации, то можно указать кодовую базу с помощью *URL*-адреса. При этом предполагается, что в указанном месте доступны указанные файлы. Можно указать конкретное место в файловой системе, где расположены требуемые для работы класс-файлы, но будем моделировать ситуацию, когда клиентская и серверная стороны приложения работают на разных компьютерах и поэтому кодовая база указана таким образом. Понятно, что при запуске приложения понадобится работающий *Web*-сервер. В данном приложении не нужно передавать параметр инициализации активируемого объекта, поэтому последний аргумент метода также `null`.

Следующим действием, можно зарегистрировать дескриптор активации в службе активации `rmid` следующим образом:

```
Calculator stub = (Calculator) Activatable.register(desc);
```

Статический метод `register()` класса `Activatable` (см. <https://docs.oracle.com/javase/8/docs/api/java/rmi/activation/Activatable.html>) регистрирует дескриптор активации и возвращает ссылку на заглушку объекта, который должен быть активирован по требованию. Эта заглушка может быть зарегистрирована в реестре обычным образом:

```
Naming.rebind("calculator", stub );
```

Приведем полный код простого класса сервера:

```
package activation;

import java.io.File;
import java.rmi.Naming;
import java.rmi.activation.*;
import java.util.Properties;

public class Server {
    public static void main( String args[] ) throws Exception
    {
        Properties props = new Properties();
```

```

String path = new File("policy").getCanonicalPath();
props.put("java.security.policy", path);

ActivationGroupDesc aGroup = new
    ActivationGroupDesc(props, null);
System.out.print("Registering activation group
descriptor...");

ActivationGroupID id =
    ActivationGroup.getSystem().registerGroup(aGroup);
System.out.println("[ OK ]");

String location = "http://localhost:8090/RMI/";
ActivationDesc desc = new ActivationDesc(id,
    RemoteCalculator.class.getName(),
    location, null);
System.out.print("Registering activation descriptor...
");
Calculator stub = (Calculator)
    Activatable.register(desc);
System.out.println("[ OK ]");
System.out.println("Obtained stub for the
SimpleCalculator");
Naming.rebind("calculator", stub );
System.out.println("Stub for SimpleCalculator bound in
registry");
    }
}

```

Создание класса клиента

Класс клиента, по смыслу, практически не отличается от реализации клиентов на прошлом занятии:

```

package activation;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
    public static void main(String[] args) {
        try {
            String name = "calculator";
            Registry registry = null;
            if (args.length > 0) {
                registry =
                    LocateRegistry.getRegistry(args[0]);
            }
        }
    }
}

```

```

    } else {
        registry =
            LocateRegistry.getRegistry("localhost");
    }

    Calculator cal =
        (Calculator) registry.lookup(name);

    int x = 12, y = 3;
    int result = cal.add(x,y);
    System.out.println("Sent: " + x + " and " + y);
    System.out.println("Received("+x+"+"+y+") = " +
        result);
    result = cal.subtract(x,y);
    System.out.println("Sent: " + x + " and " + y);
    System.out.println("Received("+x+"-"+y+") = " +
        result);
    result = cal.multiply(x,y);
    System.out.println("Sent: " + x + " and " + y);
    System.out.println("Received("+x+"*"+y+") = " +
        result);
    result = cal.divide(x,y);
    System.out.println("Sent: " + x + " and " + y);
    System.out.println("Received("+x+"/"+y+") = " +
        result);
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

Разверывание, компиляция и запуск приложения

Будем сразу моделировать ситуацию, когда серверная и клиентская части нашего распределенного приложения расположены на разных компьютерах – в нашей модели запуска в разных каталогах.

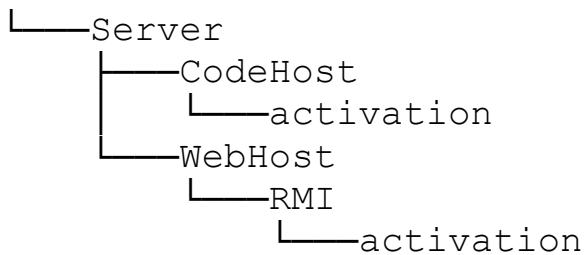
Исходные файлы классов нашего приложения могут быть разработаны и откомпилированы в любой интегрированной среде разработки, а можно их создать в простом текстовом редакторе с подсветкой синтаксиса и откомпилировать вручную – как было указано в предыдущих лекциях. (Вообще-то, пример создавался и компилировался в среде разработки, а потом класс-файлы размещались в каталогах).

Создадим такую структуру каталогов:

```

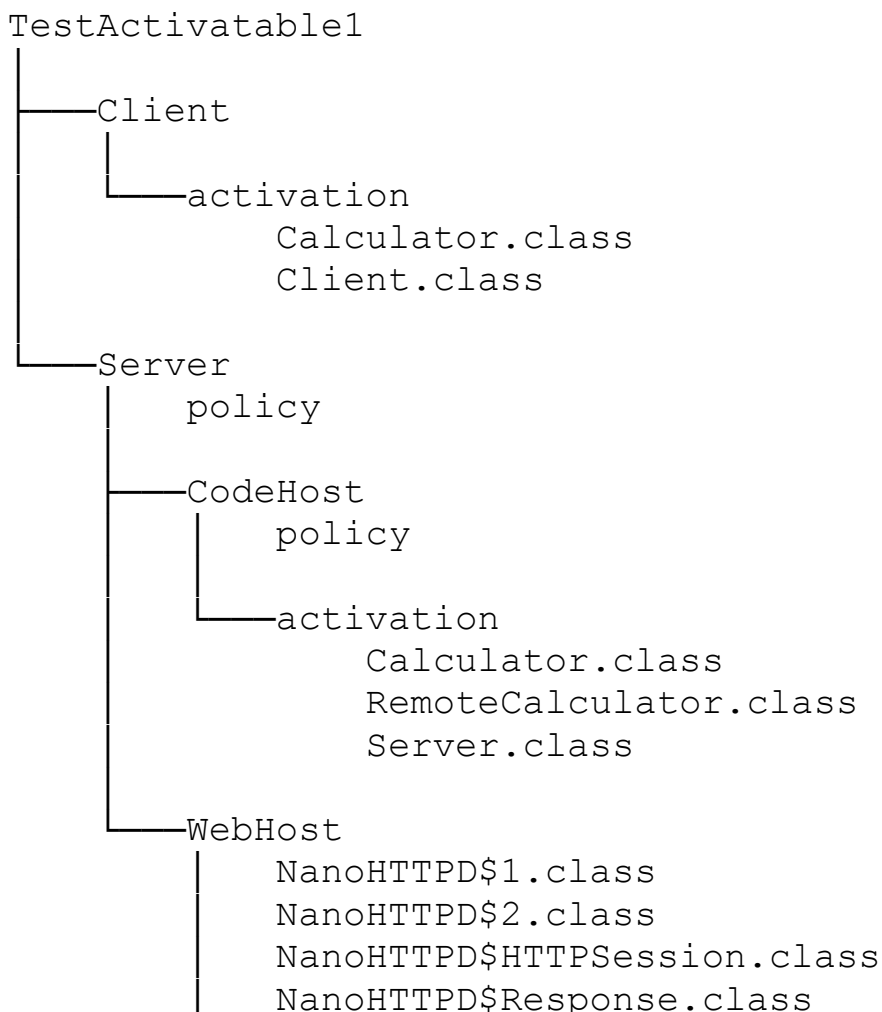
TestActivatable1
├── Client
└── activation

```



Здесь каталог `TestActivatable1` – каталог, в котором размещены каталоги серверной части – `Server` и клиентской части – `Client` нашего приложения. На серверной стороне (в каталоге `Server`) создадим каталог `CodeHost` (в нашей модели запуска этот каталог представляет компьютер, где размещен код приложения), и каталог `WebHost` (компьютер с работающим *Web*-сервером). Каталог `RMI` будет корневым каталогом *Web*-сервера, а каталог `activation` – каталог нашего пакета, в котором хранятся классы приложения.

Разнесем класс-файлы по каталогам. Приведем схему размещения файлов. Начнем по-порядку. В каталоге `Client` нашего приложения нужно разместить клиентскую часть – в каталог пакета `activation` на клиентской стороне нужно поместить класс-файлы удаленного интерфейса (`Calculator.class`) и собственно код клиента (`Client.class`).



```
    NanoHTTPD.class
    NanoHTTPD.java
  └── RMI
      └── activation
          Calculator.class
          RemoteCalculator.class
```

В каталоге **Server** размещаем файл политики безопасности **policy** – отсюда будем запускать демон активации – приложение **rmid**. В каталог **CodeHost** – серверную часть приложения – поместим файл политики безопасности **policy**, а в каталог пакета **activation** – откомпилированный файл удаленного интерфейса (**Calculator.class**), откомпилированный класс-файл класса, реализующего этот удаленный интерфейс (**RemoteCalculator.class**) и класс-файл серверной части приложения (**Server.class**).

В каталог **WebHost** помещаем файл с исходным кодом нашего **HTTP**-сервера (**NanoHTTPD.java**) и компилируем его. В каталог пакета **activation**, размещенного здесь же каталога **RMI**, помещаем файлы нужные службе реестра (класс-файл удаленного интерфейса **Calculator.class**), и службе активации (класс-файл реализации удаленного интерфейса **RemoteCalculator.class**).

После размещения файлов по каталогам приступаем к процедуре запуска нашего распределенного приложения.

Сначала нужно осуществить запуск **Web**-сервера. Для этого, находясь в каталоге **WebHost** нужно, как мы уже делали раньше, выполнить команду

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

Будет запущен наш маленький **Web**-сервер и на экран будет выведено терминальное окно с информационными сообщениями, а **Web**-сервер будет работать на порту 8090.

```

C:\Windows\system32\cmd.exe
d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ0\Server\WebHost>java -cp . NanoHTTPD -P 8090
NanoHTTPD 1.27 (C) 2001, 2005-2013 Jarno Elonen and (C) 2010 Konstantinos Togiass
(Command line options: [-p port] [-d root-dir] [--licence])

Now serving files in port 8090 from "d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ0\Server\WebHost\."
Hit Enter to stop.

```

На следующем шаге запустим службу имен – службу *RMI* реестра.

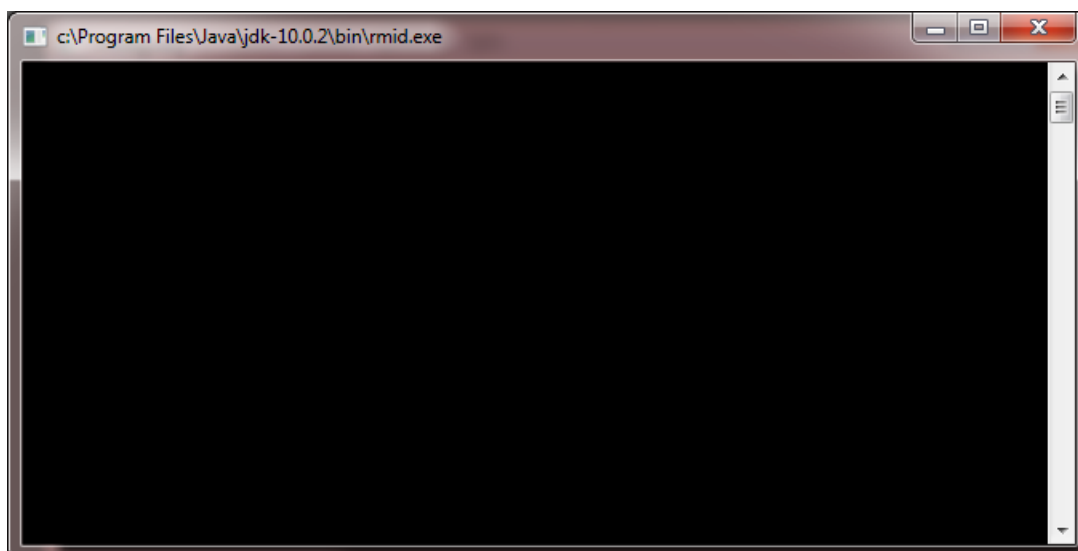


Для этого из каталога *Server* выполним команду запуска:

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

На экран будет выведено текстовое окно службы реестра. При таком способе запуска кодовая база службе реестра будет передана со стороны сервера.

Затем выполним запуск демона активации – приложение *rmid*.



Для этого, находясь в том же каталоге **Server** выполним команду:

```
start rmid -J-Djava.security.policy=policy
-C-Djava.rmi.server.useCodebaseOnly=false
```

С помощью опции **-J** определяем параметр, который передается интерпретатору *Java*, выполняющему **rmid**, а с помощью опции **-C** можно определить параметр, который передается в качестве аргумента командной строки каждому дочернему процессу (*группе активации, JVM*), создаваемому программой **rmid** во время создания этого процесса.

Затем нужно подождать некоторое время (в литературе рекомендуется подождать примерно 2 мин), пока **rmid** не закончит действия по настройке. На экран будет выведено терминальное окно, куда будут выведены сообщения удаленного объекта.

Затем запускаем серверную часть приложения. Для этого переходим в каталог сервера **CodeHost** и запускаем сервер командой:

```
java -cp . -Djava.security.policy=policy
-Djava.rmi.server.codebase=http://localhost:8090/RMI/
activation.Server
```

На экран будет выведено диалоговое окно с информационными сообщениями серверной части.


```

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\Activ0\Server\CodeHost>3.bat

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\Activ0\Server\CodeHost>java -cp . -Djava.security.policy=policy -Djava.rmi.server.codebase=http://localhost:8090/RMI/ activation.Server
Registering activation group descriptor... [ OK ]
Registering activation descriptor... [ OK ]
Obtained stub for the SimpleCalculator
Stub for SimpleCalculator bound in registry

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\Activ0\Server\CodeHost>

```

Следует отметить, что при регистрации заглушки в службе имен программе `rmiregistry` потребуется класс-файл удаленного интерфейса. Из-за сделанных настроек кодовой базы он будет загружен с *Web*-сервера. Соответствующие информационные сообщения будут выведены в терминальное окно информационных сообщений сервера.

```

C:\Windows\system32\cmd.exe

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\Activ0\Server\WebHost>java -cp . NanoHTTPD -P 8090
NanoHTTPD 1.27 (C) 2001,2005-2013 Jarno Elonen and (C) 2010 Konstantinos Togiass
(Command line options: [-p port] [-d root-dir] [--licence])

Now serving files in port 8090 from "d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\Activ0\Server\WebHost\."
Hit Enter to stop.

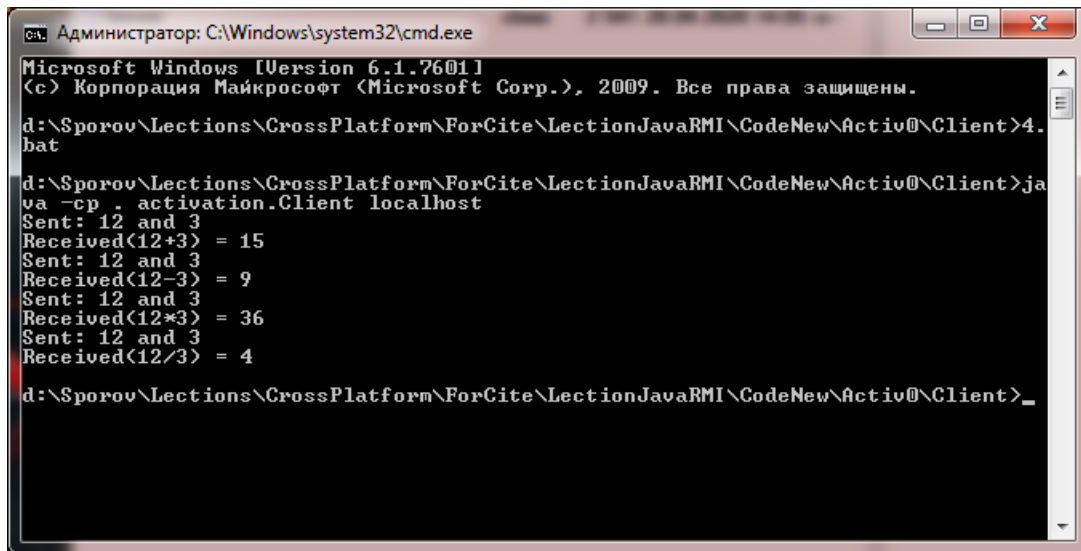
GET '/RMI/activation/Calculator.class'
  HDR: 'connection' = 'keep-alive'
  HDR: 'accept' = 'text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2'
  HDR: 'host' = 'localhost:8090'
  HDR: 'user-agent' = 'Java/10.0.2'

```

После успешного запуска серверной части можно запускать клиентскую часть распределенного приложения. Для этого переходим в каталог клиента `Client` и оттуда даем команду на запуск клиентской части:

```
java -cp . activation.Client localhost
```

Будут выполнены все соединения, сервером динамической активации создан удаленный объект, вызваны удаленные методы и на экран будет выведено терминальное окно, в котором будут отображены результаты вычислений, сделанных клиентской частью приложения.



```

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

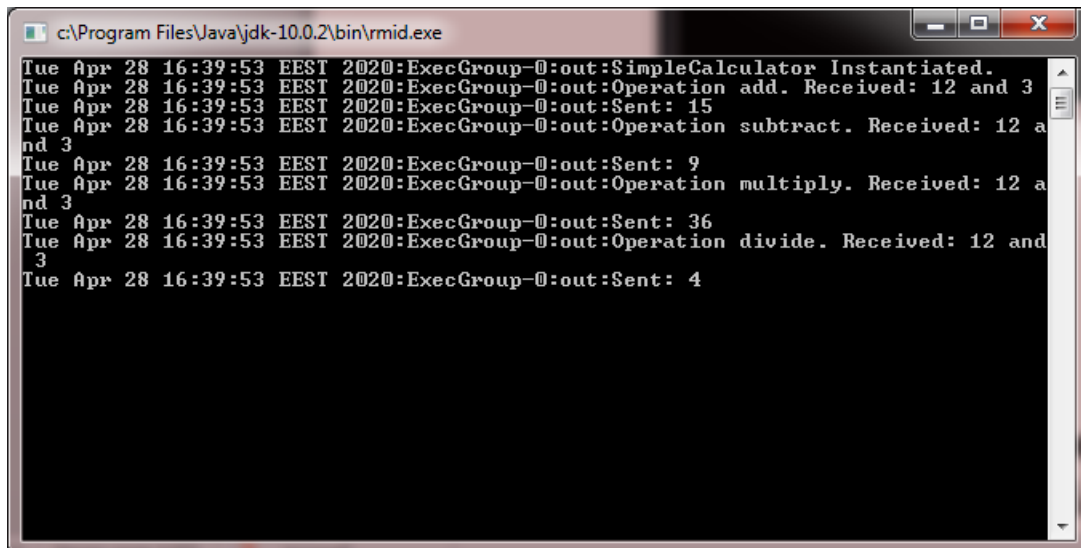
d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\Activ0\Client>4.
bat

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\Activ0\Client>ja
va -cp . activation.Client localhost
Sent: 12 and 3
Received(12+3) = 15
Sent: 12 and 3
Received(12-3) = 9
Sent: 12 and 3
Received(12*3) = 36
Sent: 12 and 3
Received(12/3) = 4

d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\Activ0\Client>_

```

В окне службы активации выводятся сообщения от активного (созданного и работающего) удаленного объекта.

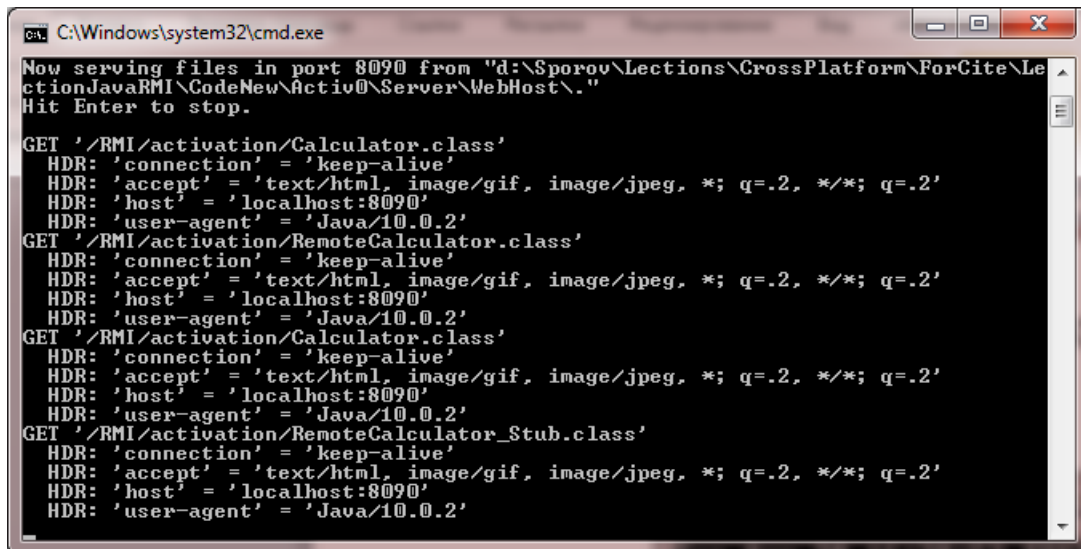


```

c:\Program Files\Java\jdk-10.0.2\bin\rmid.exe
Tue Apr 28 16:39:53 EEST 2020:ExecGroup-0:out:SimpleCalculator Instantiated.
Tue Apr 28 16:39:53 EEST 2020:ExecGroup-0:out:Operation add. Received: 12 and 3
Tue Apr 28 16:39:53 EEST 2020:ExecGroup-0:out:Sent: 15
Tue Apr 28 16:39:53 EEST 2020:ExecGroup-0:out:Operation subtract. Received: 12 a
nd 3
Tue Apr 28 16:39:53 EEST 2020:ExecGroup-0:out:Sent: 9
Tue Apr 28 16:39:53 EEST 2020:ExecGroup-0:out:Operation multiply. Received: 12 a
nd 3
Tue Apr 28 16:39:53 EEST 2020:ExecGroup-0:out:Sent: 36
Tue Apr 28 16:39:53 EEST 2020:ExecGroup-0:out:Operation divide. Received: 12 and
3
Tue Apr 28 16:39:53 EEST 2020:ExecGroup-0:out:Sent: 4

```

При этом класс-файлы, требуемые для создания удаленного объекта, будут загружены с *Web*-сервера. Соответствующие сообщения о загруженных класс-файлах будут выведены в виде информационных сообщений в терминальном окне *Web*-сервера.



```

C:\Windows\system32\cmd.exe
Now serving files in port 8090 from "d:\Sporov\Lectons\CrossPlatform\ForCite\Le
ctionJavaRMI\CodeNew\Activ0\Server\WebHost\."
Hit Enter to stop.
GET '/RMI/activation/Calculator.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/10.0.2'
GET '/RMI/activation/RemoteCalculator.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/10.0.2'
GET '/RMI/activation/Calculator.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/10.0.2'
GET '/RMI/activation/RemoteCalculator_Stub.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/10.0.2'

```

Можно еще раз запустить клиента, в окне службы динамической активации будут показаны сообщения удаленного объекта, а так как объект заново не создается, то с сервера новые класс-файлы загружены не будут.

Приложение заканчивается обычным образом – клиентское и серверное приложения будут самостоятельно закончены. Для завершения работы демона активации `rmid` в каком-то терминальном окне нужно указать команду завершения работы:

```
rmid -stop
```

Данная команда найдет работающий процесс, соответствующий программе `rmid`, и завершит его работу. В том терминальном окне, в котором работает `rmid`, будет выведено сообщение

```
activation daemon shut down
```

и демон активации закончит свою работу.

Следует отметить, что для успешной работы приложения нужно корректно указать все команды, иметь свободные порты служб имен и активации, или запустить их на других портах и иметь полностью настроенный *JDK* на компьютере.

Второй демонстрационный пример

Как мы уже обсуждали, в первом варианте распределенных приложений, выполненных в соответствии с технологией *RMI*, серверная программа используется для создания и регистрации объектов, чтобы клиенты могли выполнять на них удаленные вызовы. Однако, для некоторых задач нецелесообразно создавать большое количество удаленных объектов и заставлять их ожидать подключения, так как неясно, а будет ли клиент, которому потребуется именно этот удаленный объект. Механизм активации позволяет отложить создание объекта, таким образом, чтобы удаленный

объект создается только тогда, когда хотя бы один клиент вызывает на нем удаленный метод.

При такой структуре программы код клиента может остаться таким же, как и в первом случае: клиент также запрашивает удаленную ссылку и с помощью нее вызывает удаленный метод на удаленном объекте.

Во втором случае обычная серверная программа заменяется программой активации. Эта программа должна создать дескрипторы активации для объектов, которые будут созданы позднее, и зарегистрировать заглушки для вызова удаленных вызовов методов в службе имен. При первом вызове удаленного метода информация, содержащаяся в дескрипторе активации, используется для создания объекта.

Немного изменим код нашего «вежливого» приложения, чтобы воспользоваться преимуществами активации.

Удаленный интерфейс

В рамках рассматриваемого подхода удаленный интерфейс может остаться неизменным:

```
package activation2;

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

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

В данном интерфейсе также определен один удаленный метод, который по заданному правилу будет формировать строку приветствия.

Класс, реализующий удаленный интерфейс

Удаленный объект, который можно активировать указанным образом, должен расширять класс `Activatable` вместо класса `UnicastRemoteObject`. Конечно, он также должен имплементировать один или несколько удаленных интерфейсов. Например:

```
public class GreetingImpl extends Activatable
                               implements Greeting {
    ... ..
}
```

Поскольку создание объекта откладывается до его первого реального использования, то объект должен быть создан в соответствии со стандартной процедурой, характерной для данной технологии. Для этого необходимо предоставить конструктор, который принимает два параметра:

- идентификатор активации, который можно просто передать конструктору суперкласса;
- один объект, содержащий всю информацию, необходимую для задания начального состояния объекта; эта информация объединяется в объекте типа `MarshaledObject` (см. <https://docs.oracle.com/javase/8/docs/api/java/rmi/MarshaledObject.html>)

.

Если для указания начального состояния объекта необходимо задать несколько параметров, то их нужно упаковать в один объект. Для этого можно использовать, например, массив `Object[]` или `ArrayList`, или что-то еще.

При создании дескриптора активации, объект `MarshaledObject` создается на основе информации о состоянии объекта следующим образом:

```
MarshaledObject<T> param =
    new MarshaledObject<T>(constructionInfo);
```

В конструкторе класса, имплементирующего удаленный интерфейс, можно воспользоваться методом `get`, вызванным на объекте класса `MarshaledObject` для получения десериализованной информации об объекте.

```
T constructionInfo = param.get();
```

Для того, чтобы продемонстрировать процедуру активации с передачей параметра, определяющего начальное состояние удаленного объекта, модифицируем класс `GreetingImpl` так, чтобы эта информация была целочисленным значением, определяющим количество людей в очереди (для демонстрационных целей начальное значение будет не 0, а 3). Эта информация упаковывается в объект `MarshaledObject` и распаковывается в конструкторе удаленного объекта.

```
public GreetingImpl(ActivationID id,
    MarshaledObject<Integer> data) throws
    ActivationException,
    IOException, ClassNotFoundException {
    super(id, 0);
    num = data.get();
    System.out.println("Greeting implementation constructed");
}
```

В конструкторе сначала вызывается конструктор суперкласса, которому в качестве первого параметра передается идентификатор активации `id`, а в качестве второго параметра передается 0, тем самым предписывается, что система *RMI* должна самостоятельно назначить подходящий номер порта для слушателя.

Последним действием этот конструктор выводит на экран информационное сообщение, о том, что удаленный объект активирован по требованию.

Как и в случае обычного распределенного *RMI* приложения класс, имплементирующий удаленный интерфейс не обязан всегда расширять класс *Activatable*. Вместо этого можно выполнить явную операцию экспорта, сохранив соответствующую *Remote* ссылку или в этом конструкторе, или в коде сервера, с помощью вызова статического метода:

`Activatable.exportObject(this, id, 0)`

Приведем полный код класса, имплементирующего удаленный интерфейс:

```
package activation2;

import java.io.IOException;
import java.rmi.MarshalledObject;
import java.rmi.RemoteException;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationException;
import java.rmi.activation.ActivationID;

public class GreetingImpl extends Activatable
    implements Greeting {

    private int num;

    public GreetingImpl(ActivationID id,
        MarshalledObject<Integer> data) throws
        ActivationException, IOException,
        ClassNotFoundException {

        super(id, 0);
        num = data.get();
        System.out.println("Greeting implementation
constructed");
    }

    @Override
    public String greet(String name) throws RemoteException {
        System.out.println("Obtain request for " + name);
        return "Hello, " + name + "! Your number is " +
            (++num);
    }
}
```

Активация удаленного объекта

Теперь напишем исходный код класса, реализующего программу активации. Сначала вам нужно определить группу активации. Группа активации описывает общие параметры, необходимые для запуска виртуальной машины, которая будет содержать удаленные объекты. Самым важным параметром является политика безопасности.

Создайте дескриптор группы активации с помощью такого кода:

```
Properties props = new Properties();
props.put("java.security.policy", "/path/to/server.policy");
ActivationGroupDesc group = new ActivationGroupDesc(props, null);
```

Второй параметр конструктора дескриптора группы активации описывает специальные параметры командной строки. Для нашего примера они не нужны, поэтому и передаем `null`.

Следующим шагом следует создать идентификатор группы с помощью такого фрагмента кода:

```
ActivationGroupID id =
    ActivationGroup.getSystem().registerGroup(group);
```

После этого можно приступить к созданию дескрипторов активации. Для каждого объекта, который должен быть создан по требованию, нужна такая информация:

- идентификатор группы активации для виртуальной машины, в которой должен быть создан удаленный объект;
- полное имя класса удаленного объекта (например, `"MyClassImpl"` или `"com.mycompany.MyClassImpl"`), которое удобно получить при помощи рефлексии;
- строка, сформированная по правилам *URL*, указывает место, откуда будут загружены класс-файлы удаленных объектов. Это должен быть базовый *URL*, который не включает путь к пакетам;
- маршаллизованная информация, необходимая для задания начального состояния объекта при создании.

Например:

```
MarshaledObject param = new MarshaledObject(constructionInfo);
ActivationDesc desc = new ActivationDesc(id, "MyClassImpl",
    "http://myserver.com/download/", param);
```

Затем следует передать созданный дескриптор активации статическому методу `Activatable.register`. Он возвращает объект заданного типа, который реализует заданные удаленные интерфейсы класса реализации. Этот объект затем привязывается в службе имен:


```
MyClassInterf stub = (MyClassInterf) Activatable.register(desc);
Naming.bind("rmi://localhost:6789/myobject", stub);
```

В отличие от серверных программ, разработанных на прошлой и позапрошлой лекциях, программа активации завершает работу сразу после регистрации дескриптора активации и привязки заглушки к службе имен. Удаленные объекты создаются только при первом вызове удаленного метода.

Рассмотрим полный код программы активации:

```
package activation2;

import java.io.File;
import java.io.IOException;
import java.rmi.MarshalledObject;
import java.rmi.Naming;
import java.rmi.activation.*;
import java.util.Properties;

public class GreetingActivator {
    public static void main(String[] args) throws
        ActivationException, IOException {
        System.out.println("Constructing activation
descriptors...");

        Properties props = new Properties();
        // use the server.policy file in the current directory
        props.put("java.security.policy",
            new File("server.policy").getCanonicalPath());
        ActivationGroupDesc group =
            new ActivationGroupDesc(props, null);
        ActivationGroupID id =
            ActivationGroup.getSystem().registerGroup(group);

        MarshalledObject<Integer> param = new
            MarshalledObject<Integer>(Integer.parseInt("3"));

        String codebase = "http://localhost:8090/RMI/";
        ActivationDesc desc = new ActivationDesc(id,
            GreetingImpl.class.getName(),
            codebase, param);

        Greeting greeting = (Greeting)
            Activatable.register(desc);

        System.out.println("Binding activable implementation
```



```

to registry...");
    Naming.rebind("rmi://localhost:6789/greeting",
                  greeting);
    System.out.println("Exiting...");
}
}

```

Клиентская часть приложения

Клиентскую часть приложения можно оставить без изменений. Приведем исходный код клиентского класса:

```

package activation2;

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

public class GreetingClient {
    public static void main(String[] args) {
        String host = "localhost";
        int port = 6789;
        try {
            Greeting stub = (Greeting)
                Naming.lookup("rmi://" + host +
                             ":" + port + "/greeting");
            System.out.println(stub.greet("Vasisyalij"));
        } catch (NotBoundException e) {
            e.printStackTrace();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}

```

Разверывание, компиляция и запуск приложения

Как и для всех прошлых примеров, будем моделировать ситуацию, когда серверная и клиентская части нашего распределенного приложения расположены на разных компьютерах – в нашей модели запуска они будут размещены в разных каталогах. Для удобства, все каталоги приложения разместим в общую папку TestActivatable2.

Пропустим этап компиляции исходных файлов нашего приложения. Все действия аналогичны там, что были сделаны ранее. В данном примере все исходные файлы были написаны и откомпилированы в интегрированной

среде разработки *IntelliJ Idea*, а затем класс-файлы были размещены в соответствующих каталогах.

Создадим такую структуру каталогов для нашего распределенного приложения:

```

TestActivatable2
├── client
│   └── activation2
├── rmi
│   └── log
├── server
│   └── activation2
└── Web
    └── RMI
        └── activation2

```

Здесь каталог `TestActivatable2` – это общий каталог приложения, в котором размещены каталоги серверной части – `server` и клиентской части – `client` нашего приложения. К серверной части приложения относятся и каталог `rmi` – именно из него будем запускать службы реестра и активации. В этом каталоге указан каталог `log` – его создавать не нужно, это служебный каталог, который будет автоматически создан при начале работы демона активации. Каталог `Web` представляет компьютер с работающим *Web*-сервером. В этом каталоге создадим каталог `RMI`, который будет корневым каталогом *Web*-сервера. Осталось только в каталогах `server`, `client` и `RMI` создать каталог `activation2` – каталог пакета, в котором хранятся класс-файлы нашего приложения.

Разнесем класс-файлы по каталогам. Приведем схему размещения файлов. Начнем по-порядку. В каталоге `client` нашего приложения нужно разместить клиентскую часть – в каталог пакета `activation2` на клиентской стороне нужно поместить класс-файлы удаленного интерфейса (`Greeting.class`) и собственно код клиента (`GreetingClient.class`). Кроме этого в каталоге `client` размещен пакетный файл `client_start.bat`, в котором указана команда запуска клиентской части приложения. В каталоге `rmi` разместим файл с политикой безопасности `server.policy`. Содержимое этого файла:

```

grant
{
    permission java.security.AllPermission;
};

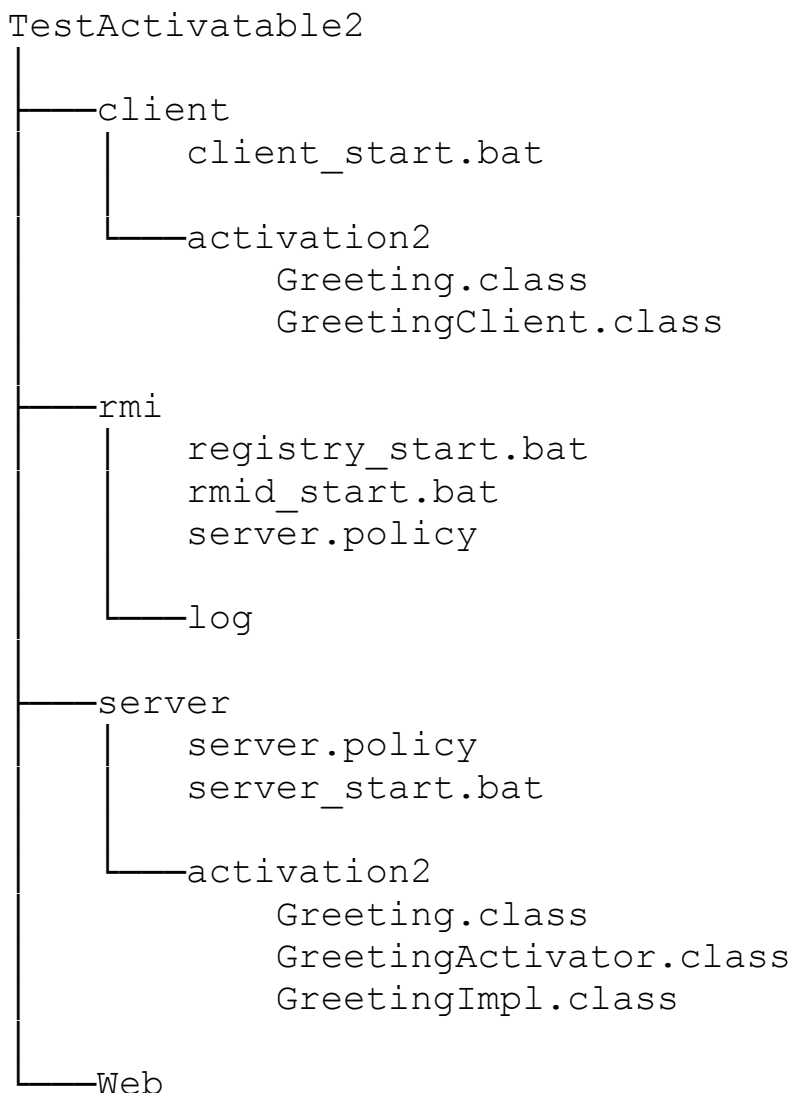
```

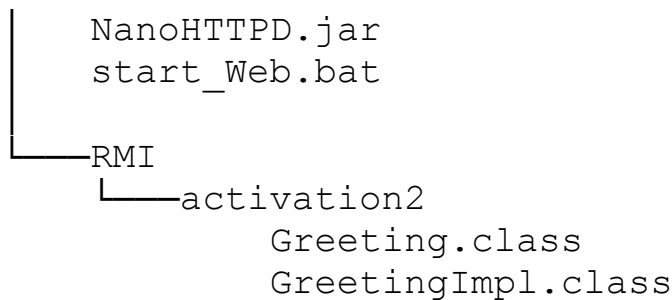
Именно из этого каталога будем запускать службу реестра и службу активации, и для упрощения запуска можно разместить здесь пакетные файлы `registry_start.bat` и `rmid_start.bat` с соответствующими командами запуска.

В каталоге `server` в каталоге пакета `activation2` нужно поместить класс-файлы удаленного интерфейса (`Greeting.class`), откомпилированный класс-файл класса, реализующего этот удаленный интерфейс (`GreetingImpl.class`) и класс-файл серверной части приложения (`GreetingImpl.class`). Кроме этого, в каталоге `server` должен находиться файл политики безопасности `server.policy` с указанным выше содержанием, необходимый для работы приложения. Для удобства запуска серверной части приложения в этом каталоге можно разместить пакетный файл `server_start.bat` с командой запуска.

В каталог `Web` можно поместить `jar`-файл с класс-файлами нашего `HTTP`-сервера (как объединять класс-файлы в `jar`-файл мы уже рассматривали на одной из прошлых лекций). Кроме того, как раньше, в этот же каталог можно поместить пакетный файл `start_Web.bat`, содержащий команду запуска `HTTP`-сервера. В корневом каталоге сервера `RMI` в каталог пакета `activation2`, размещенного здесь же, помещаем файлы нужные службе реестра (класс-файл удаленного интерфейса `Greeting.class`), и службе активации (класс-файл реализации удаленного интерфейса `GreetingImpl.class`).

Получившаяся структура каталогов с файлами:



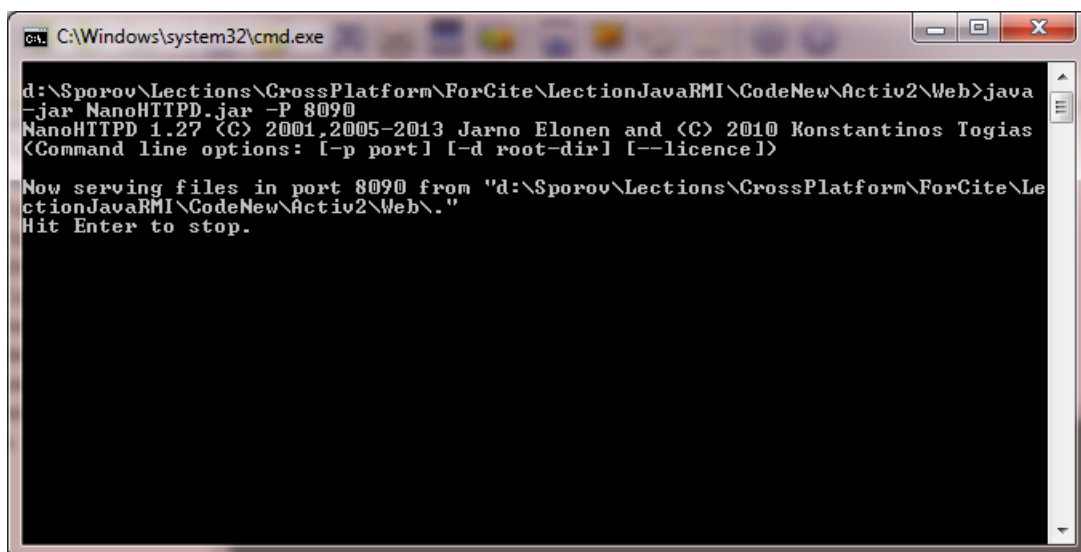


После размещения файлов по каталогам приступаем к процедуре запуска нашего распределенного приложения.

Сначала запустим *Web*-сервер. Для этого, находясь в каталоге *Web* нужно выполнить команду

```
java -jar NanoHTTPD.jar -P 8090
```

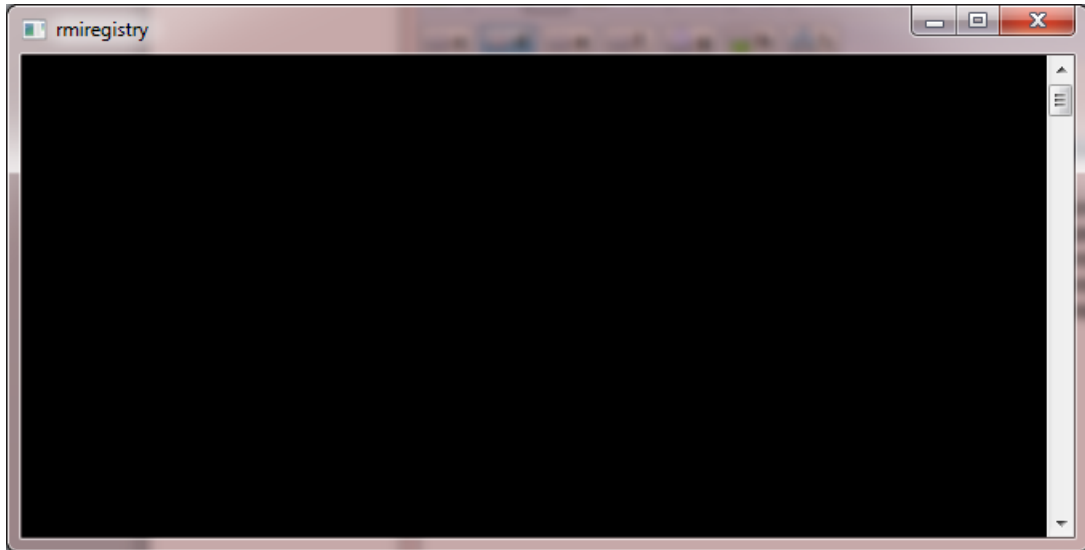
Будет запущен наш маленький *Web*-сервер и на экран будет выведено терминальное окно с информационными сообщениями, а *Web*-сервер будет работать на порту 8090.



На следующем шаге запустим службу имен – службу *RMI* реестра. Для этого из каталога *rmi* выполним команду запуска:

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

На экран будет выведено текстовое окно службы реестра. При таком способе запуска кодовая база службе реестра будет передана со стороны сервера. По другому службе реестра класс-файлы доступны не будут.

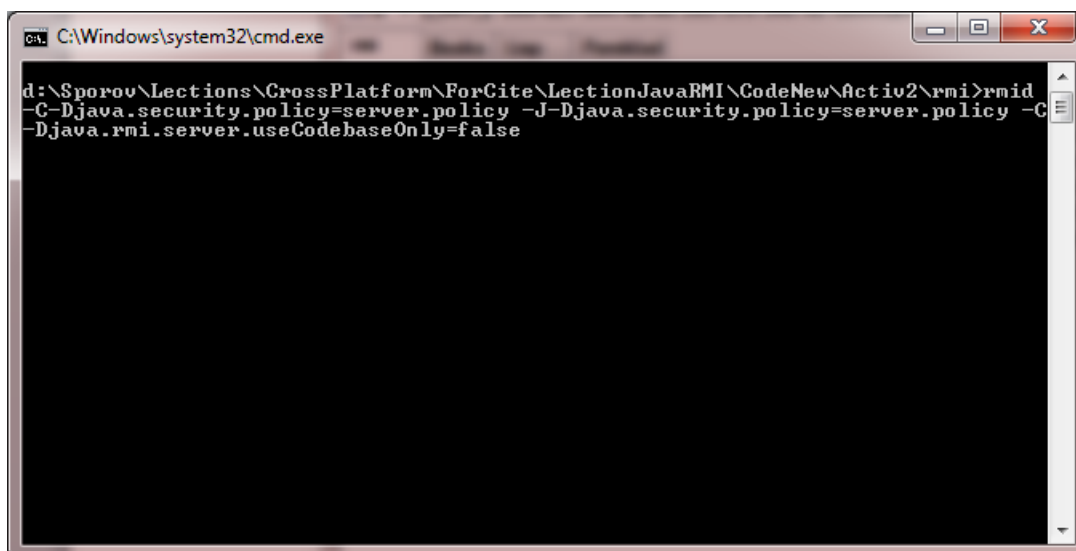


Затем выполним запуск демона активации – приложение `rmid`. Для этого, находясь в том же каталоге `rmi` выполним команду:

```
rmid -C-Djava.security.policy=server.policy  
-J-Djava.security.policy=server.policy  
-C-Djava.rmi.server.useCodebaseOnly=false
```

Программа `rmid` отслеживает запросы на активацию и активирует объекты в отдельной виртуальной машине. Для запуска виртуальной машины программе `rmid` необходимы определенные разрешения. Они указаны в файле политики, который указан в команде запуска. Параметр `-J` используется для передачи параметра виртуальной машине, на которой запущен демон активации.

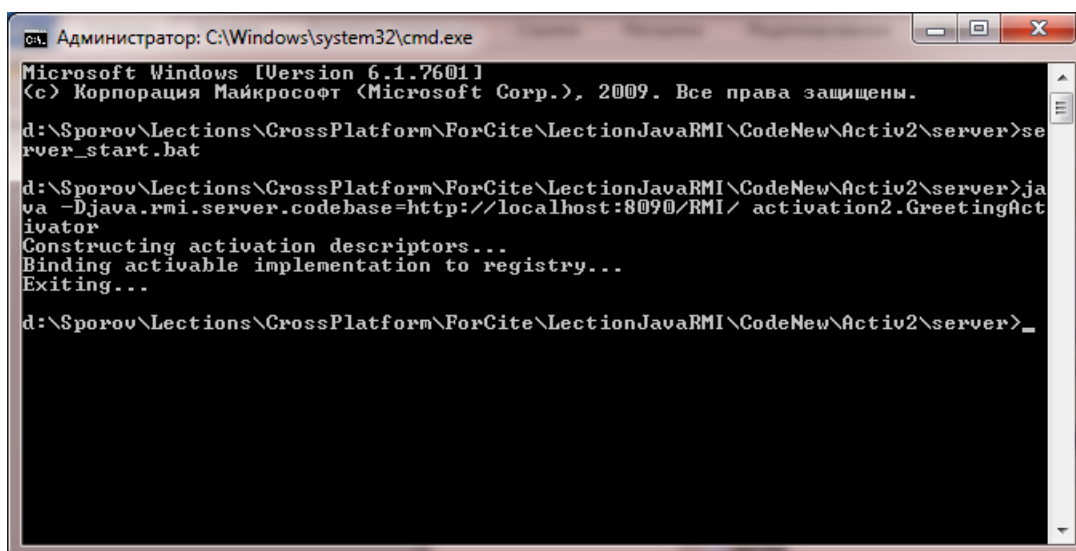
Затем нужно подождать некоторое время (в литературе рекомендуется подождать примерно 2 мин), пока `rmid` не закончит действия по настройке. На экран будет выведено терминальное окно, куда будут выведены сообщения удаленного объекта.



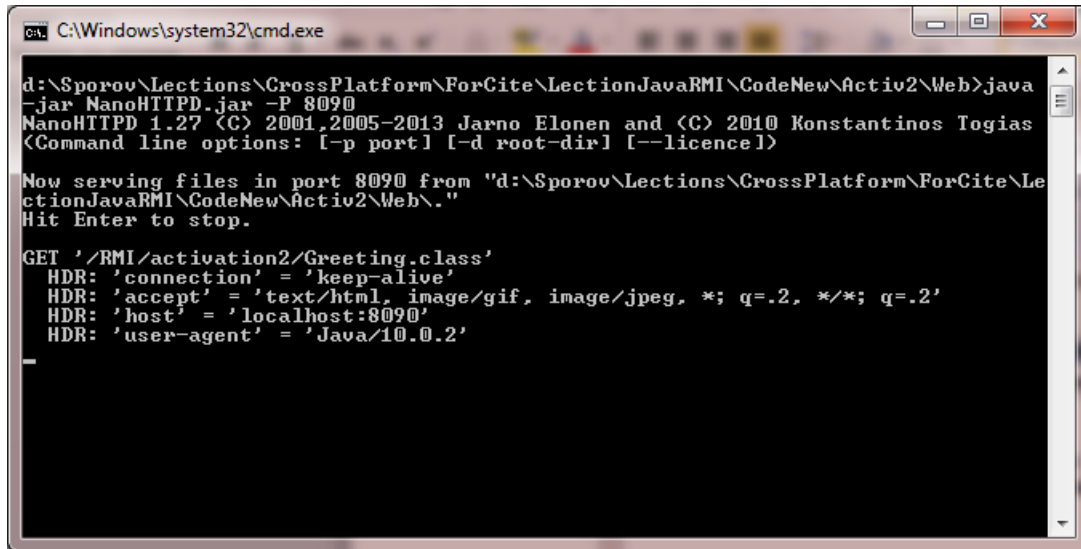
Затем запускаем серверную часть приложения. Для этого переходим в каталог сервера `server` и запускаем сервер командой:

```
java -Djava.rmi.server.codebase=http://localhost:8090/RMI/
activation2.GreetingActivator
```

Запущенная программа активации выведет информацию в терминальное окно и завершает работу сразу после выполнения действий по регистрации дескриптора активации в службе активации и созданных заглушек в службе имен. При запуске этой чатси приложения в строке запуска нужно указать параметр, определяющий кодовую базу, несмотря на то, что она уже указана в конструкторе дескриптора активации. Эта указанная в конструкторе информация используется только демоном активации *RMI*, а служба *RMI* реестра по-прежнему (в соответствии с нашими параметрами запуска) получает кодовую базу из команды запуска серверной части.



При регистрации заглушки в службе реестра программе `rmiregistry` потребуется класс-файл удаленного интерфейса. Из-за сделанных настроек кодовой базы он будет загружен с *Web*-сервера. Соответствующие информационные сообщения будут выведены в терминальное окно информационных сообщений сервера.



```

C:\Windows\system32\cmd.exe
d:\Sporov\Lectons\CrossPlatform\FoCite\LectonJavaRMI\CodeNew\Activ2\Web>java
-jar NanoHTTPD.jar -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\Lectons\CrossPlatform\FoCite\Le
ctonJavaRMI\CodeNew\Activ2\Web\."
Hit Enter to stop.

GET '/RMI/activation2/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 успешного запуска серверной части можно запускать клиентскую часть распределенного приложения. Для этого переходим в каталог клиента `client` и отсюда даем команду на запуск клиентской части:

```
java -cp . activation2.GreetingClient
```

Будут выполнены все соединения, сервером динамической активации создан удаленный объект, вызваны удаленные методы и на экран будет выведено терминальное окно, в котором будут отображены результаты вычислений, сделанных клиентской частью приложения.

```

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\Activ2\client>cl
ient_start.bat

d:\Sporov\Lectons\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\Activ2\client>ja
va -cp . -Djava.security.policy=server.policy activation2.GreetingClient
Hello, Vasisyalij! Your number is 4

d:\Sporov\Lectons\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\Activ2\client>

```

Web-сервер обеспечивает загрузку классов, необходимых для активации объекта. Соответствующие сообщения о загруженных класс-файлах будут выведены в виде информационных сообщений в терминальном окне *Web*-сервера.

```

C:\Windows\system32\cmd.exe
Now serving files in port 8090 from "d:\Sporov\Lectons\CrossPlatform\ForCite\Le
ctionJavaRMI\CodeNew\Activ2\Web\."
Hit Enter to stop.

GET '/RMI/activation2/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'
GET '/RMI/activation2/GreetingImpl.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'
GET '/RMI/activation2/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'
GET '/RMI/activation2/GreetingImpl_Stub.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'

```

В информационном окне демона активации выводятся сообщения об активации объекта и информация о запросе на вызов удаленного метода на удаленном объекте.


```

C:\Windows\system32\cmd.exe

d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ2\rmi>rmi
-C-Djava.security.policy=server.policy -J-Djava.security.policy=server.policy -C
-Djava.rmi.server.useCodebaseOnly=false
Wed Apr 29 10:43:13 EEST 2020:ExecGroup-0:out:Greeting implementation constructe
d
Wed Apr 29 10:43:13 EEST 2020:ExecGroup-0:out:Obtain request for Uasisyalij

```

Можно выполнить вызовы удаленных методов еще несколько раз.

```

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

d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ2\client>cl
ient_start.bat

d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ2\client>ja
va -cp . -Djava.security.policy=server.policy activation2.GreetingClient
Hello, Uasisyalij! Your number is 4

d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ2\client>cl
ient_start.bat

d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ2\client>ja
va -cp . -Djava.security.policy=server.policy activation2.GreetingClient
Hello, Uasisyalij! Your number is 5

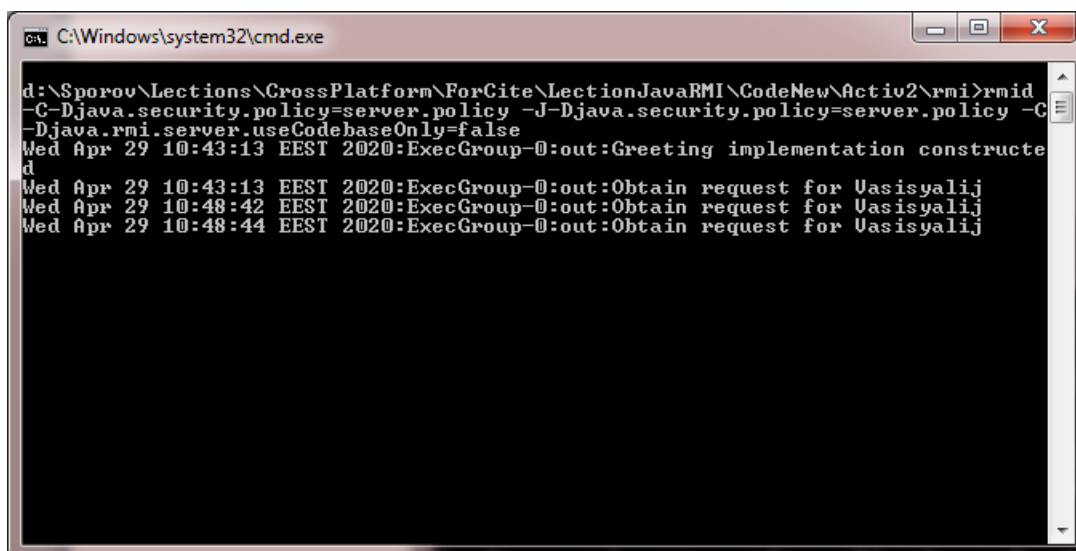
d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ2\client>cl
ient_start.bat

d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ2\client>ja
va -cp . -Djava.security.policy=server.policy activation2.GreetingClient
Hello, Uasisyalij! Your number is 6

d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ2\client>

```

Такие повторные вызовы метода не приводят к созданию нового (повторной активации удаленного объекта) – объект уже находится в активном состоянии и это можно отследить, просматривая сообщения в информационном окне демона активации: есть информационные сообщения о запросах на вызов даленного метода, но нет сообщений об повторной активации объекта.



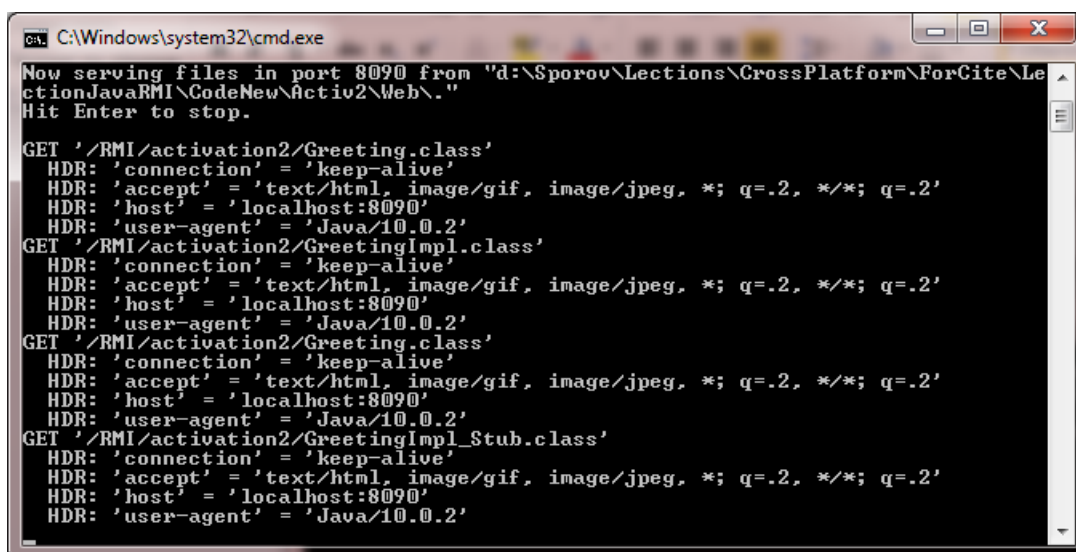
```

C:\Windows\system32\cmd.exe

d:\Sporov\Lectons\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\Activ2\rmi>rmid
-C-Djava.security.policy=server.policy -J-Djava.security.policy=server.policy -C
-Djava.rmi.server.useCodebaseOnly=false
Wed Apr 29 10:43:13 EEST 2020:ExecGroup-0:out:Greeting implementation constructe
d
Wed Apr 29 10:43:13 EEST 2020:ExecGroup-0:out:Obtain request for Vasisyalij
Wed Apr 29 10:48:42 EEST 2020:ExecGroup-0:out:Obtain request for Vasisyalij
Wed Apr 29 10:48:44 EEST 2020:ExecGroup-0:out:Obtain request for Vasisyalij

```

Ну и как следствие, при таком повторном вызове удаленных методов на активном удаленном объекте на *Web*-сервер не поступают новые запросы на предоставление класс-файлов.



```

C:\Windows\system32\cmd.exe

Now serving files in port 8090 from "d:\Sporov\Lectons\CrossPlatform\ForCite\Le
ctionJavaRMI\CodeNew\Activ2\Web\"
Hit Enter to stop.

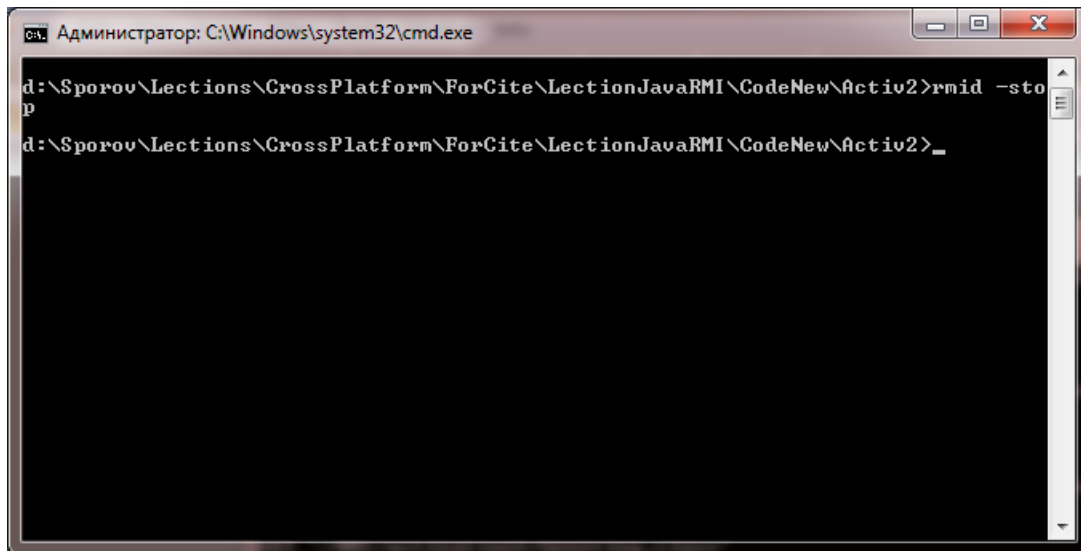
GET /RMI/activation2/Greeting.class'
HDR: 'connection' = 'keep-alive'
HDR: 'accept' = 'text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2'
HDR: 'host' = 'localhost:8090'
HDR: 'user-agent' = 'Java/10.0.2'
GET /RMI/activation2/GreetingImpl.class'
HDR: 'connection' = 'keep-alive'
HDR: 'accept' = 'text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2'
HDR: 'host' = 'localhost:8090'
HDR: 'user-agent' = 'Java/10.0.2'
GET /RMI/activation2/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/10.0.2'
GET /RMI/activation2/GreetingImpl_Stub.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/10.0.2'

```

Завершаем работу удаленного приложения: клиентская часть выполнила все задания и завершила работу; серверная часть приложения, связанная с активацией, тоже сразу закончила работу. Завершаем работу *Web*-сервера, нажав на клавишу <Enter> в его терминальном информационном окне. Сервер заканчивает работу и закрывает терминальное окно.

Завершаем работу демона активации. На серверном компьютере в любом терминальном окне вводим команду останова демона активации:

`rmid - stop`



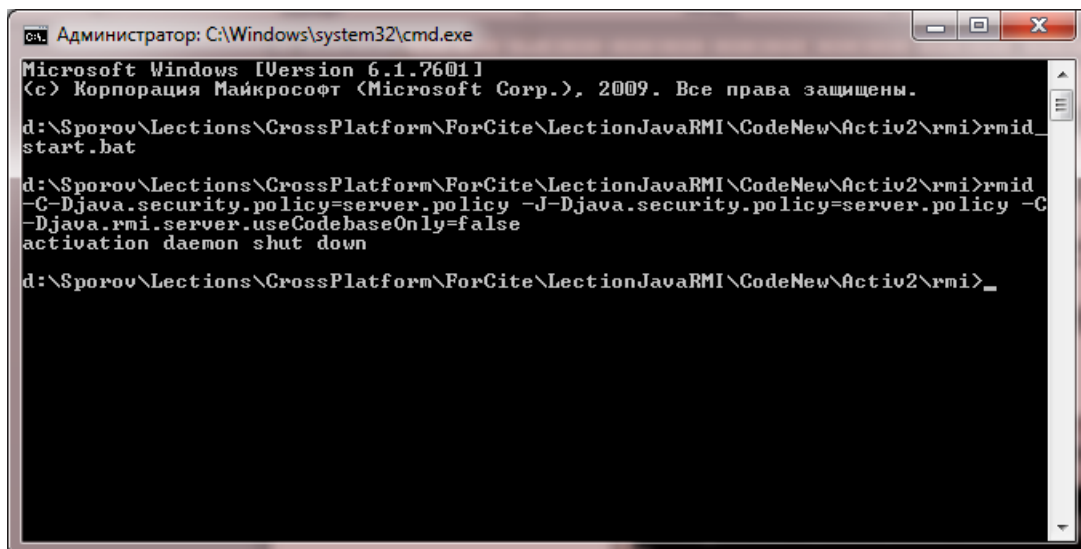
```

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

d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ2>rmdir -stop
d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ2>_

```

Будет найден процесс, соответствующий демону активации и его работа будет завершена. Если он был запущен из отдельно созданного терминального окна, то в это окно будет выведено информационное сообщение о завершении работы службы активации:



```

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

Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ2\rmi>rmdir
start.bat

d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ2\rmi>rmdir
-C-Djava.security.policy=server.policy -J-Djava.security.policy=server.policy -C
-Djava.rmi.server.useCodebaseOnly=false
activation daemon shut down

d:\Sporov\Lectiions\CrossPlatform\ForCite\LectiionJavaRMI\CodeNew\Activ2\rmi>_

```

Ну а службу реестра удаленных объектов завершаем средствами операционной системы, например, просто закрыв соответствующее терминальное окно.