

Тема: Remote Method Invocation (Part 1)

План занятия:

1. Введение

- Основы RMI
 - Серверная часть
 - Клиентская часть
 - Реестр объектов
- Основные этапы создания RMI приложения
 - Определение удаленного интерфейса
 - Имплементация удаленного интерфейса
 - Создание серверной части
 - Создание клиентской части
 - Этап компиляции
 - Запуск приложения

2. Основные интерфейсы и классы Java RMI

3. Простой пример распределенного приложения

- Определение удаленного интерфейса
- Серверная часть приложения
 - Реализация удаленного интерфейса
 - Реализация удаленного метода
 - Создание конструктора
 - Создание сервера RMI
 - Создание удаленного объекта
 - Экспорт объекта
 - Регистрация заглушки
- Создание RMI клиента

4. Запуск распределенного приложения

- Все классы в одном пакете
 - Компиляция приложения
 - Запуск службы реестра
 - Запуск серверной части приложения
 - Запуск клиентской части приложения
- Классы в разных пакетах (возможно на разных хостах)
 - Запуск программы «в старом стиле»
 - Запуск программы «в новом стиле»
 - Особенности запуска RMI приложений в современной Java
- Варианты запуска RMI приложения
 - Способ 1
 - Способ 2
 - Способ 3

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

Введение

Мы уже рассмотрели различные варианты сетевых программ, реализованных при помощи сокетов (поточковых, датаграмных и многоадресных). Наличие такого понятия как «сокет» хотя и упрощает создание сетевых программ, но все-таки неочень сильно. Даже для такого простого задания, как передача информации с одного хоста на другой нужно разработать протокол передачи и написать достаточно много стандартного кода. Для более сложных сетевых задач нужно выполнить еще больше стандартных, рутинных действий, которые чреваты ошибками.

Для *Java* содержит набор классов и интерфейсов для организации объектно-ориентированной сетевой работы. Предоставляется механизм, пользуясь которым программист, разрабатывающий клиент / серверные программы, мог бы формировать вызов методов, не заботясь о передаче данных по сети, получении и разборе ответа. Этот подход известен как *Remote Method Invocation* (*Вызов удаленных методов*). Рассмотрим основные возможности такого подхода.

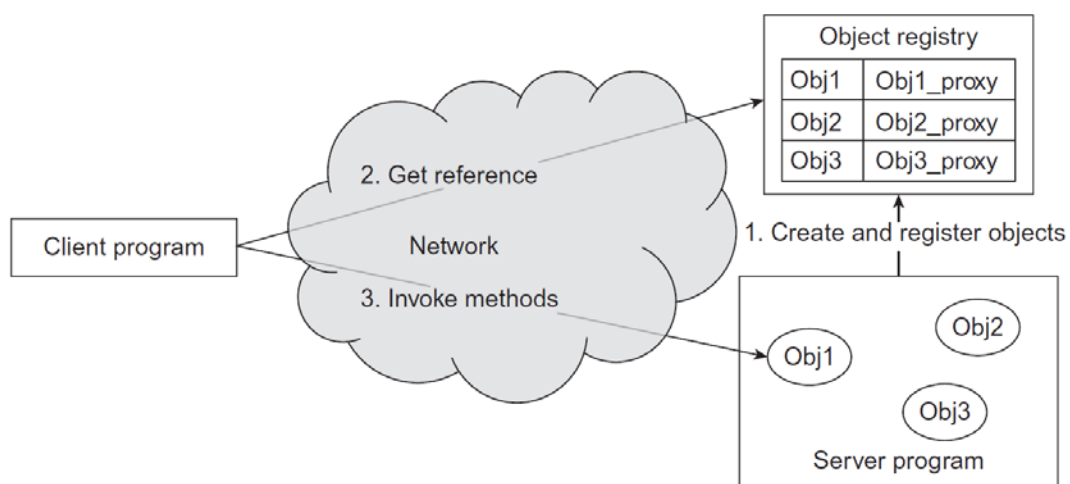
Основы RMI

Java Remote Method Invocation (RMI) - это объектно-ориентированная версия удаленного вызова процедур (*RPC*). С помощью такого подхода можно без явного использования сокетов вызвать методы для объекта, который существует в другом адресном пространстве или на том же компьютере, или на другом компьютере, подключенном к исходному через сеть. Таким образом, с помощью этого подхода, можно организовать взаимодействие объектов, расположенных на разных компьютерах. Это достаточно простая, элегантная, но в то же время мощная технология, которая используется для вызова удаленных методов в случаях, когда на всех концах взаимодействия установлена *Java*.

Первоначально архитектура *RMI* была разработана с учетом таких основных целей:

- позволить программистам разрабатывать распределенные программы *Java* с тем же синтаксисом и семантикой, что и у нераспределенных программ.
- создать распределенную, безопасную и надежную объектную модель, которая естественно соответствует языку программирования *Java* и его объектной модели.

Рассмотрим схему построения приложения в соответствии с технологией *RMI*.



В рамках этой модели взаимодействие между клиентами и сервером в этой модели происходит без явного использования сокетов (Они конечно используются, но незаметно для программиста). Технически, то, что

происходит, не является вызовом метода в смысле ООП, но внешне все выглядит похоже.

В структуре RMI приложения можно выделить три стороны: серверная часть, клиентская часть и реестр объектов (*object registry*).

Серверная часть

Обычно, задача этой части программы состоит в создании удаленного объекта (*remote object*), который будет нужен для вызова метода. Этот объект является обычным объектом, за исключением того, что его класс реализует специальный интерфейс *Java RMI*. После того, как удаленный объект создан, он экспортируется и регистрируется в отдельном приложении, называемом реестром объектов. Кратко рассмотрим назначение и обязанности этих частей.

Клиентская часть

Клиентская часть приложения обычно сначала связывается с реестром объектов, для того, чтобы получить ссылку на удаленный объект (дескриптор объекта) с заданным именем. После этого, используя полученную ссылку (дескриптор), клиентская часть может вызывать методы на удаленном объекте так, как будто объект хранится в собственном адресном пространстве клиента. Технология *RMI* сама, без явного участия программиста, обрабатывает детали связи между клиентом и сервером (внутренне используя сокет) и передает информацию в нужном направлении. Эта сложная процедура связи полностью скрыта для клиентских и серверных приложений.

Реестр объектов (Object Registry)

Реестр объектов по своему смыслу похож на таблицу. Каждая запись таблицы отображает имя объекта на его прокси (*proxy*), который называется заглушка (*stub*). Серверная часть приложения регистрирует заглушку с указанием имени в реестре объектов. После того, как заглушка будет успешно зарегистрирована в реестре объектов, объект становится доступным для использования другими объектами. Сразу после этого клиенты могут получить ссылку на удаленный объект из этого реестра (дескриптор объекта, фактически заглушку) и могут вызывать методы на этом объекте.

Основные этапы создания RMI приложения

Обычно, чтобы как-то структурировать процесс разработки и запуска *RMI* приложений, можно выделить следующие шаги:

- определение удаленного интерфейса (*remote interface*);
- создание класса, реализующего удаленный интерфейс;

- создать серверную часть приложения, которая создает, экспортирует и регистрирует удаленный объект в реестре объектов;
- создать клиентскую часть приложения, которая получит ссылку на удаленный объект и будет использовать ее для организации вызова метода на удаленном объекте;
- откомпилировать исходные файлы *Java*;
- в правильной последовательности запустить части удаленного приложения.

Рассмотрим немного подробнее основные этапы создания и работы распределенного приложения *RMI*.

Определение удаленного интерфейса

При написании распределенного приложения нужно определить интерфейс, который должен содержать заголовки методов, которые серверный объект хочет сделать доступными для удаленного вызова клиента. Так как, с технической точки зрения, вызов метода на удаленном объекте выполняется не так, как вызов метода на локальном объекте, интерфейс для удаленного объекта должен удовлетворять следующим требованиям:

- удаленный интерфейс (*remote interface*) должен обязательно быть *public*.
- удаленный интерфейс должен прямо или косвенно расширять (*extends*) интерфейс *java.rmi.Remote*. См. страницу документации на сайте производителя:

<https://docs.oracle.com/javase/8/docs/api/java/rmi/Remote.html>.

Интерфейс *Remote* является маркерным интерфейсом и не определяет ни одного метода *public interface java.rmi.Remote { }*. Удаленный интерфейс расширяет этот интерфейс *Remote* только для указания особого статуса своих методов – только они могут быть вызваны удаленно.

- каждый метод удаленного интерфейса должен объявить, что он может выбросить исключение *java.rmi.RemoteException*. Дело в том, что при работе с удаленным объектом могут возникнуть проблемы с этим объектом не связанные напрямую – например проблема с сетевым подключением и т.д. Именно поэтому каждый удаленный метод должен предоставить дополнительное исключение *RemoteException*, для того, чтобы разработчик мог требуемым образом обработать эти проблемы. См. документацию по исключению на сайте производителя: <https://docs.oracle.com/javase/8/docs/api/java/rmi/RemoteException.html>.

Имплементация удаленного интерфейса

На следующем шаге создания распределенного приложения нужно создать конкретный класс, реализующий один или несколько таких удаленных интерфейсов. Эти классы, кроме удаленных, могут реализовывать и обычные интерфейсы. Классы также могут содержать другие открытые методы, но то, что не содержится в удаленных интерфейсах, может быть вызвано только локально. Кроме того, на этом этапе должны быть реализованы все классы, которые указаны в качестве параметров или возвращаемых значений для удаленных методов. При необходимости можно создать более чем одну реализацию удаленного интерфейса. Вызывающая сторона не обязана знать о базовой реализации.

Создание серверной части

На следующем этапе нужно создать серверную часть приложения. Основные задачи, которые должна выполнить эта часть: создать экземпляр удаленного объекта, а затем зарегистрировать его в *RMI* реестре с заданным именем. Эта стадия обычно называется *object deployment* (развертывание объекта). Есть несколько стандартных способов зарегистрировать объект в *RMI* реестре. С некоторыми основными мы познакомимся на этом занятии чуть позже.

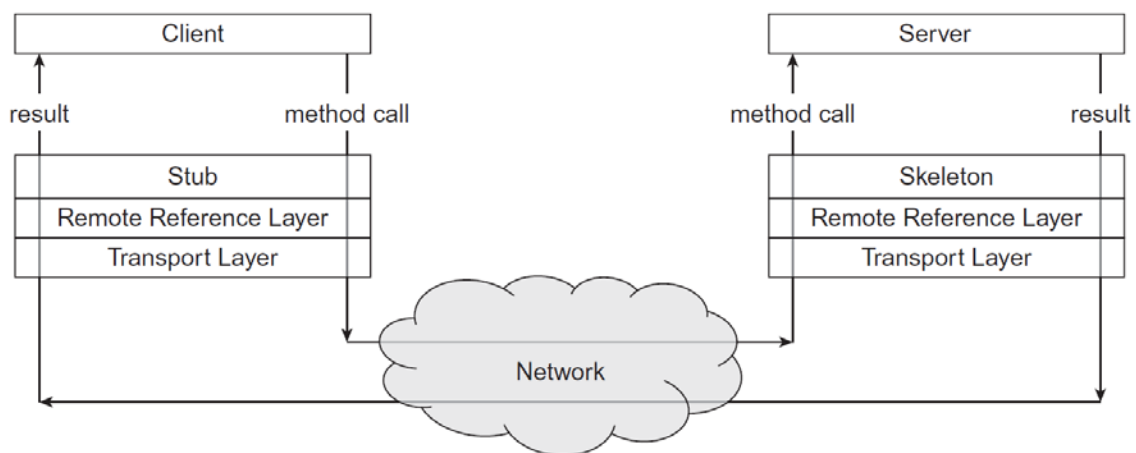
Создание клиентской части

Сразу после, или параллельно с серверной частью, можно приступить к созданию клиентской части приложения. Основная задача этой части – связаться со службой *RMI* реестра, получить по зарегистрированному имени дескриптор удаленного объекта и с его помощью организовать вызов удаленных методов. Существуют разные способы получения ссылки на удаленный объект. Некоторые из них мы разберем на этом занятии чуть позже.

Этап компиляции

После того, как готов исходный классов, наступает этап компиляции частей приложения. Для компиляции всех исходных файлов, включая файлы интерфейсов и других вспомогательных классов, используется обычный компилятор *Java* (*javac*). Для успешной работы технологии *RMI* необходимо создать и «заставить» работать заглушку (*stub*) и скелетон (*skeleton*) – вспомогательные служебные классы (на самом деле это *прокси*). Нужно обратить внимание на то, что в ранних версиях *Java* (до *Java 5.0*) разработчикам *RMI* приложений приходилось создавать эти служебные классы *RMI* на отдельном этапе компиляции с использованием специального

компилятора *rmic*, входящего в стандартную поставку *JDK*. Начиная с версии *Java 5.0* и дальше, это шаг больше не является обязательным. Стабы и скелетоны создаются динамически, по мере необходимости, при работе *RMI* приложения (вспомним динамические прокси из темы рефлексия). Схематически рассмотрим назначение и сценарий работы заглушки и скелетона (см. Рис):



- Начнем рассмотрение схемы работы с поведения клиента. Когда клиент вызывает удаленный метод, то на локальной заглушке происходит вызов соответствующего метода.
- Заглушка (*stub*) упаковывает (маршализирует, *marshals*) необходимую для вызова метода информацию (имя метода, параметры, передаваемые методы во время вызова и т.д.) и посылает собранную информацию на серверную часть приложения скелетону (*skeleton*). Сам процесс сбора данных и преобразования их в стандартный формат для передачи по сети называется *маршаллингом* (*marshaling*). Заглушка (*stub*) знает все необходимое для установления связи по сети со скелетоном (*IP* адрес, порт).
- Скелетон (*skeleton*), расположенный на серверной стороне приложения, после получения этой информации, распаковывает ее, восстанавливает объекты – параметры метода (*анмаршаллинг*, *unmarshals*) и вызывает указанный метод на реальном объекте. Затем упаковывает полученный результат и отправляет его (если результат есть) обратно заглушке.
- Заглушка получает, распаковывает информацию и передает результат клиенту.

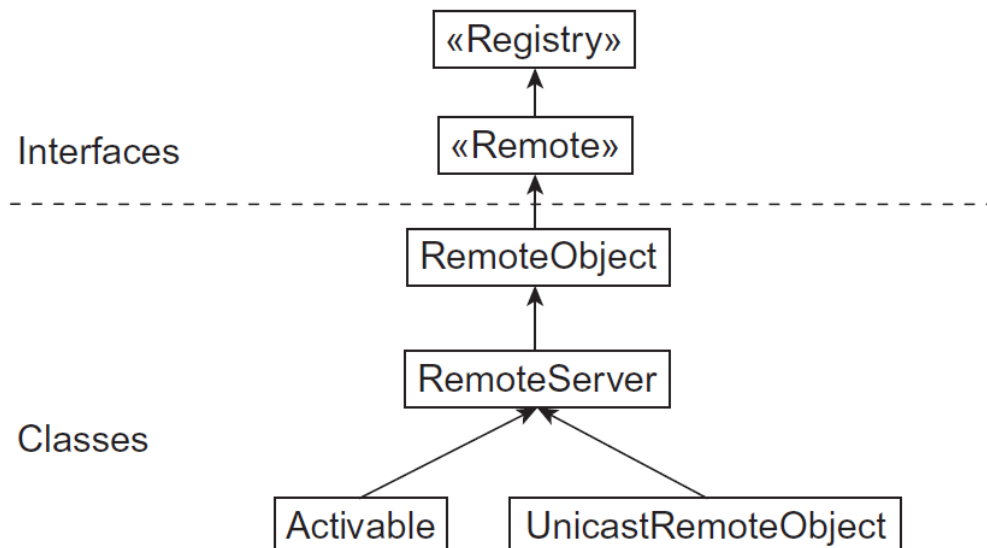
Запуск приложения

Запуск приложения *RMI* – достаточно сложная и громоздкая процедура. Рассмотрим основные этапы, а подробности разберем на примерах. Сначала

необходимо запустить *RMI* службу реестра объектов. В *Java* реестр объектов можно запустить с помощью команды `rmiregistry`. Кроме того, реестр объектов можно создать динамически. После запуска реестра запускается серверная часть приложения. После того, как серверная часть запущена и успешно работает, запускается клиентская часть приложения.

Основные интерфейсы и классы *Java RMI*

Технология *Java RMI* реализована в виде интерфейсов и классов в пакете `java.rmi`. При разработке *RMI* приложений этот пакет обычно включается в исходные файлы. Технология *Java RMI* скрывает практически все особенности распределенного приложения и обеспечивает единообразный способ доступа к объектам (как к распределенным объектам, так и к локальным). Рассмотрим иерархию основных классов и интерфейсов технологии *Java RMI*:



В последующих примерах рассмотрим использование этих классов и интерфейсов, а для полного знакомства с возможностями нужно смотреть документацию:

<https://docs.oracle.com/javase/8/docs/api/java/rmi/registry/Registry.html>

<https://docs.oracle.com/javase/8/docs/api/java/rmi/Remote.html>

<https://docs.oracle.com/javase/8/docs/api/java/rmi/server/RemoteObject.html>

<https://docs.oracle.com/javase/8/docs/api/java/rmi/server/RemoteServer.html>

<https://docs.oracle.com/javase/8/docs/api/java/rmi/server/UnicastRemoteObject.html>

Простой пример распределенного приложения

Рассмотрим пример простого распределенного приложения. Понятно, что настолько простые примеры на практике не используются. Главное

назначение этого примера – разобрать основные этапы разработки приложения *RMI*.

Рассмотрим распределенное приложение, которое будет «здороваться» с пользователем. Сначала напишем самую простую версию приложения, а затем будем усложнять ее. Разработку можно вести в любой среде разработки, только нужно помнить – есть три географически разные стороны (группы разработчиков): комитет, определяющий протокол взаимодействия, группа, разрабатывающая серверную часть приложения и группа, разрабатывающая клиентскую часть приложения. И это особенность желательно сохранить. Но, для начала, чтобы постепенно разобраться с особенностями распределенного приложения, все разместим в одном пакете. А потом сделаем как надо.

Определение удаленного интерфейса

Сначала серверная и клиентская части приложения должны согласовать протокол. Этот процесс согласования выражается в интерфейсе, который обычно разрабатывается поставщиком услуг.

В нашем приложении удаленный объект предоставляет только один метод, который принимает строку в качестве аргумента, формирует строку приветствия, которую возвращает в качестве результата.

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

```
package all_in_one;

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

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

Интерфейс *Greeting*, по сути, определяет вид удаленного объекта с точки зрения клиента. С помощью этого интерфейса заинтересованные в удаленном объекте программы (клиентские программы / пользователи сервисов) могут узнать подробности сервисов (заголовки методов), которые предоставляет объект. Таким образом, с помощью интерфейса *Greeting* сервер и клиенты согласовывают правила работы (согласовывают протокол).

Этот интерфейс расширяет интерфейс *java.rmi.Remote* и, именно поэтому, считается удаленным интерфейсом. Любой объект, класс которого реализует этот интерфейс, является удаленным объектом и метод, определенный в удаленном интерфейсе, может быть вызван из другой *JVM*.

Интерфейс описывает только один метод `greet()`. Это метод будут использовать удаленные программы для получения строки приветствия от сервера. Поскольку этот метод будет вызван удаленно, то его вызов может закончиться неудачей из-за проблем, связанных со связью, протоколом или сервером. Поэтому, указывается, что во время работы метода может быть выброшено исключение `java.rmi.RemoteException`. может быть вызван этим методом во время его выполнения. Это проверяемое исключение, поэтому программа, вызывающая этот метод, должна или перехватить и явно обработать это исключение, либо передать его дальше.

Удаленный интерфейс должен быть объявлен с модификатором `public`. Дело в том, что удаленный интерфейс, скорее всего, будет использоваться классами, которые не принадлежат к тому же пакету, где объявлен интерфейс. Следовательно, чтобы сделать интерфейс доступным для всех классов приложения, он должен быть объявлен как `public`.

В рамках подхода *RMI* для передачи объектов по сети используется технология бинарной сериализации (*binary serialization*). Таким образом, объекты, которые передаются через сеть, должны быть *сериализуемыми*, т.е. класс объекта должен реализовывать либо интерфейс `java.io.Serializable`, либо `java.io.Externalizable`. Мы уже подробно разбирали эту тему и на лекции, и на практике. В нашем приложении клиент передает серверному объекту один объект `String`, как аргумент, для осуществления вызова метода `greet()`, а удаленный серверный объект передает клиенту другой объект типа `String` как результат работы метода. Тип `String` относится к встроенным типам *Java* (будет гарантированно поддерживаться на всех сторонах приложения), которые реализуют стандартный интерфейс сериализации. Таким образом, для нашего приложения проблема передачи объектов по сети решена.

Серверная часть приложения

Перейдем к созданию серверной части приложения. Задача сервера – создать удаленный объект и зарегистрировать его в службе реестра *RMI*, таким образом, сделав его доступным для клиентов.

Созданный удаленный интерфейс просто определяет обязанности удаленного объекта, ничего не говоря о том, как же конкретно эти обязанности будут реализованы. Реализация находится в классе, реализующим удаленный интерфейс. В таком классе, представляющем удаленный объект, обычно требуется:

- объявить, что класс реализует хотя бы один удаленный интерфейс;
- обеспечить реализацию всех методов, указанных в удаленном интерфейсе;

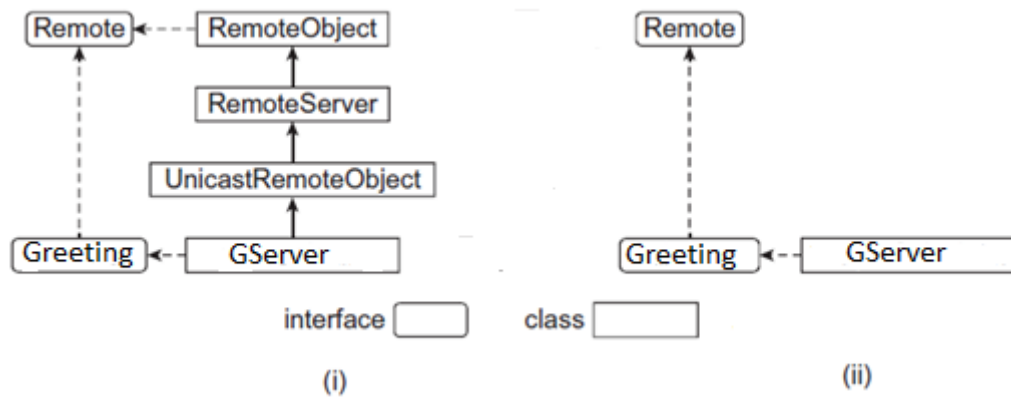
- определить достаточное количество конструкторов;
- предоставить служебные методы, которые понадобятся для работы с объектом (его настройки) как с локальным серверным объектом, и которые не будут доступны удаленным клиентам.

1. Реализация удаленного интерфейса

В принципе, есть несколько способов создать класс, реализующий удаленный интерфейс (в нашем приложении это будет класс `GServer`). Самый простой способ – это сделать так, чтобы класс, реализующий удаленный интерфейс, был наследником или класса `java.rmi.server.UnicastRemoteObject` (<https://docs.oracle.com/javase/8/docs/api/java/rmi/server/UnicastRemoteObject.html>), или класса `javax.rmi.PortableRemoteObject` (<https://docs.oracle.com/javase/8/docs/api/javax/rmi/PortableRemoteObject.html>), см. левую часть рис. ниже, (i)).

Класс `UnicastRemoteObject` рекомендуется использовать в качестве суперкласса при реализации удаленного объекта в рамках технологии *RMI*. Класс `PortableRemoteObject` должен использоваться только в программировании *RMI-IIOP*. Если используется класс `PortableRemoteObject`, то можно во время выполнения переключаться между транспортными протоколами *JRMP* и *IIOP*. см. https://docs.oracle.com/javase/6/docs/technotes/guides/rmi-iiop/rmi_iiop_pg.html.

В этом случае очень легко сделать объект удаленным. Дело в том, что объекты такого класса (наследника `UnicastRemoteObject`) автоматически экспортируются в конструкторе суперкласса (или `UnicastRemoteObject` или `PortableRemoteObject`) при их создании. Таким образом, отпадает необходимость в процедуре явного экспорта объектов. Однако, это не только преимущество, но и проблема: класс, реализующий удаленный интерфейс, не может расширять какой-либо другой класс, кроме указанных, поскольку *Java* не поддерживает множественное наследование классов. Таким образом, непосредственное расширение классов (`UnicastRemoteObject` или `PortableRemoteObject`) не всегда возможно. Поэтому в данном примере мы так делать не будем, а чуть позже рассмотрим этот способ создания удаленного объекта.



Для того, чтобы избежать такой проблемы, можно воспользоваться другим подходом. В рамках второй схемы класс, реализующий удаленный интерфейс, не расширяет классы `UnicastRemoteObject` или `PortableRemoteObject` (см. правая часть рис. выше (ii)). Это дает возможность классу реализации интерфейса расширять другие, требуемые в данной ситуации, классы. Однако объекты, созданных таким образом классов реализации, необходимо явно экспортировать в удаленную форму. Для этого класс `UnicastRemoteObject` предоставляет несколько перегруженных версий статического метода `exportObject()` для явного экспорта удаленных объектов (см., например, <https://docs.oracle.com/javase/8/docs/api/java/rmi/server/UnicastRemoteObject.html#exportObject-java.rmi.Remote-int->). Данные методы экспортируют удаленный объект, делая его доступным для приема входящих вызовов, используя, например, определенный порт. Для сетевой работы неявно используется `ServerSocket`, автоматически созданный средствами класса `RMISocketFactory`. В данном случае требуется совсем немного дополнительного кода, но при этом появляется возможность расширять те классы, которые необходимы для реализации логики приложения.

В нашем примере класса, реализующий удаленный интерфейс, создан с применением этого подхода. Рассмотрим код класса и обсудим основные моменты.

```
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);
        System.out.println("Stub");
        System.out.println(stub);
        //Registry registry = LocateRegistry.getRegistry();
        Registry registry = LocateRegistry.getRegistry(port);
        System.out.println("Registry:");
        System.out.println(registry);
        String name = "Greet";
        registry.rebind(name, stub);
        System.out.println("The server object is ready for clients");
    } catch (RemoteException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}

```

Как уже было сказано, класс реализации может имплементировать произвольное количество удаленных интерфейсов. Кроме того, класс реализации также может расширять любой другой класс, реализующий удаленный интерфейс.

Реализация удаленного метода

Класс, имплементирующий удаленный интерфейс, должен предоставлять определение для всех методов, указанных в удаленном интерфейсе. Для нашего класса реализации GServer нужно определить единственный метод `greet()`.

Метода `greet()` реализован очень просто. Он просто возвращает сформированную строку приветствия, которая включает параметр, переданный методу во время вызова и порядковый номер обратившегося клиента. В принципе, в метод можно поместить метод `System.out.println()` для отслеживания работы метода.

Нужно обратить внимание на то, что метод `greet()` не обязан указывать, что он может выбросить исключение `RemoteException`. Дело в том, что непосредственно в теле метода исключение это типа, а также другие проверяемые исключения не генерируются.

Если класс, реализующий удаленный интерфейс не предоставит определения всех методов интерфейса, то он становится абстрактным классом, и это должно быть указано в виде соответствующего модификатора в заголовке класса. Иначе компилятор сообщит об это ошибкой.

Класс, реализующий удаленный интерфейс, может содержать методы, не указанные в удаленном интерфейсе. В этом случае такие методы могут быть вызваны только локально (то есть только в той виртуальной машине, на которой запущен сервер) и не могут быть вызваны удаленно.

В удаленных методах аргументы и возвращаемые значения могут быть любого типа, допустимого в *Java*, включая типы пользователя. Для таких типов единственным существенным требованием является то, что все они должны реализовывать интерфейс `java.io.Serializable` (или `java.io.Externalizable`). Про сериализацию мы уже много узнали ранее.

Рассмотрим основные правила, которые применяются при передаче объектов удаленному методу (вообще-то все то же применимо и к возврату объектов из метода).

Когда *локальный* объект (*local object*) передается удаленному методу в качестве аргумента, то формальный параметр метода ссылается на *временный локальный* объект, который является точной копией фактического объекта. Таким образом, если вызывается удаленный метод и ему передается объект – параметр, то реальный вызов метода обрабатывает локальную «копию» объекта, а не фактический объект, который был передан в качестве аргумента. И любые изменения состояния этого локального объекта в удаленном методе отражаются только в локальной копии у получателя, а не в исходном объекте у вызывающего. Аналогично, любые изменения состояния исходного объекта на вызывающей стороне не отражаются в копии объекта на стороне получателя. При этом важно, что бы определение класса объекта – фактического параметра существовало в *JVM* на стороне удаленного объекта.

Если же объект, переданный удаленному методу в качестве параметра, сам является удаленным объектом, то формальный параметр при реальном

вызове метода ссылается на *локальный временный объект (local temporary object)*, который является *прокси (proxy)* фактического объекта. В этом случае вызов метода с формальным параметром приводит к вызову аналогичного метода для локального объекта прокси, который, в свою очередь, перенаправляет информацию о вызове метода реальному объекту. Подробное описание этой процедуры рассмотрим на этой лекции позже.

Создание конструктора

Наличие явного конструктора для класса, реализующего удаленный интерфейс, не является обязательным, если этот класс реализации не является наследником класса `java.rmi.server.UnicastRemoteObject` (или `javax.rmi.PortableRemoteObject`). Однако, если класс, реализующий удаленный интерфейс, расширяет любой из этих двух классов, то явный конструктор является обязательным. Дело в том, что если класс реализации расширяет класс `java.rmi.server.UnicastRemoteObject` или `javax.rmi.PortableRemoteObject`, то объекты класса реализации автоматически экспортируются при создании конструктором суперкласса. Поскольку во время этого экспорта конструктор суперкласса потенциально может выбросить исключение `java.rmi.RemoteException`, то просто необходимо определить конструктор, для которого указано, что может быть выброшено исключение `RemoteException`, даже если конструктор больше вообще ничего не делает. Если этого не сделать, то компилятор *Java* выдает сообщение об ошибке.

Поскольку для нашего примера класс `GServer`, имплементирующий удаленный интерфейс, не расширяет ни один из этих двух классов, то, вообще говоря, конструктор не обязателен. В нашем примере конструктор указан, так как требуется явная инициализация поля `numberOfVisitors` для каждого вновь созданного экземпляра класса.

Создание сервера RMI

Задачи, характерные для серверной стороны удаленного приложения, в нашей простой демонстрационной программе на *Java*, содержатся в методе `main()`, класса, имплементирующего удаленный интерфейс. В принципе, можно было создать отдельный класс, выполняющий аналогичные действия. В любом случае, нужно выполнить следующие шаги:

- создать экземпляр класса, реализующего удаленный интерфейс;
- экспортировать этот объект, чтобы он мог получать запрос от клиента на вызов удаленного метода;
- зарегистрировать удаленный объект в *RMI* реестре, чтобы удаленный клиент мог получить удаленную ссылку на этот объект.

Создание удаленного объекта

Экземпляр класса, реализующего удаленный интерфейс, создается точно так же, как и обычный объект. Пример создания экземпляра класса `GServer`.

```
GServer server = new GServer();
```

Экспорт объекта

Объект, созданный в подразделе выше, является обычным объектом *Java* и еще не может обрабатывать запросы на вызов методов, поступающие от удаленной программы. Чтобы добавить эту возможность, объект должен быть экспортирован (в «удаленный объект»). Экспорт объекта в основном означает то, что возвращается удаленная ссылка на объект, с помощью которой можно выполнять вызовы его методов от удаленных клиентов. Так как наш класс, реализующий удаленный интерфейс, не является наследником класса `UnicastRemoteObject` или `PortableRemoteObject`, то мы должны явно экспортировать объект, используя статический метод `exportObject()` класса `UnicastRemoteObject` (или `PortableRemoteObject`), следующим образом:

```
Greeting stub = (Greeting)
    UnicastRemoteObject.exportObject(server, 0);
```

Первый аргумент метода `exportObject()` - это тот объект, который нужно экспортировать. Второй аргумент – это целое число (`int`), определяющее *TCP* порт, который будет использован для прослушивания входящих запросов на удаленный вызов метода на удаленном объекте. Для этого параметра рекомендуется указывать значение 0 (*ноль*). Это значит, что порт будет автоматически выбран во время выполнения подсистемой *RMI* или базовой операционной системой. Но, при желании, можно указать конкретный порт, который будет использоваться для прослушивания. Если метод `exportObject()` отработает успешно, то будет возвращена удаленная ссылка на наш объект `GServer` (динамический прокси – аналогичный тому, который мы рассматривали), с помощью которой можно обрабатывать удаленные входящие вызовы.

Попробуем разобраться в функциональности метода `exportObject()` немного подробнее. Более глубокое понимание этой архитектуры позволит программистам разрабатывать достаточно сложные сетевые приложения.

Самое главное, что делает метод `exportObject()`, так это создание прокси (*proxy*) для серверного объекта (удаленного объекта) и подготовка его к работе на стороне сервера. При этом серверный объект передается прокси. Такой прокси, в рамках технологии *RMI*, еще называется *скелетон* (*skeleton*), и он обязательно имеет ссылку на серверный (удаленный) объект. В основном, задачи, которые должен решить *скелетон*, состоят в следующем:

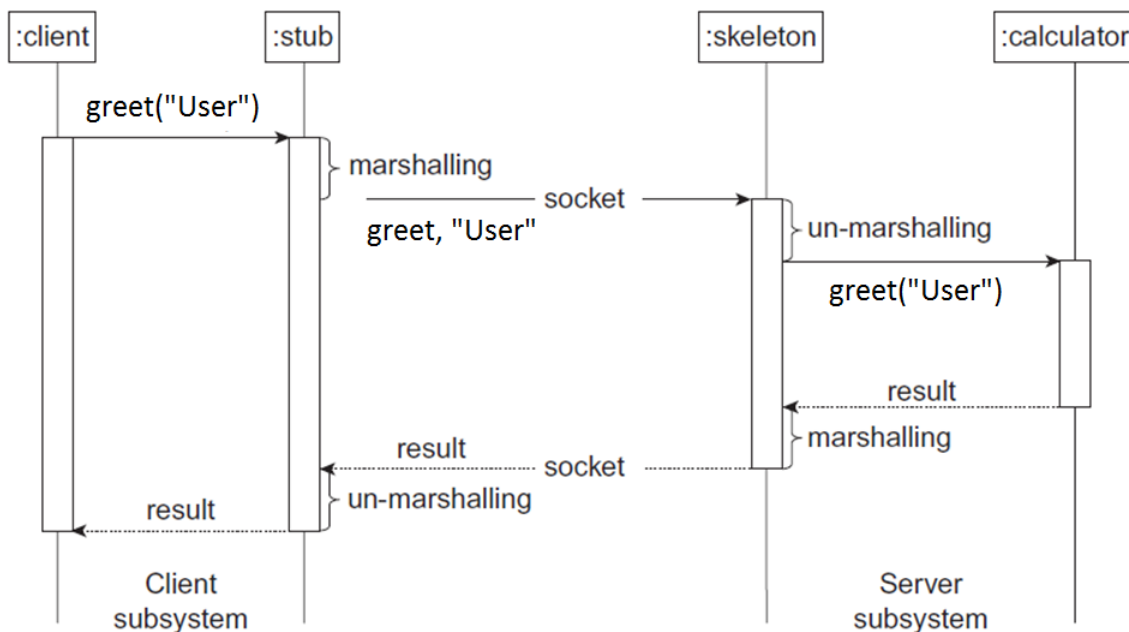
Он создает серверный сокет *TCP* (объект `ServerSocket`) и прослушивает входящие запросы на подключение к тому порту *TCP*, который был указан во втором аргументе метода `exportObject()`. После установления соединения *ServerSocket* от имени нашего объекта `server` ожидает входящего удаленного вызова метода.

Каждый раз, когда к этому скелетону (*skeleton*) поступает (в упакованном формате) новый запрос на вызов метода, скелетон распаковывает запрос,

получает имя метода и аргументы, которые переданы по сети для организации вызова. Эта процедура распаковки обычно называется *ан-маршаллинг* (*un-marshalling*). Следует обратить внимание на то, что удаленный объект *server* и его скелетон принадлежат одной и той же *JVM*, а скелетон содержит ссылку на этот локальный объект *server*. Таким образом, скелетон может вызвать нужный метод на реальном удаленном объекте (*server*) и получить результат. Затем скелетон упаковывает результат (выполняет процедуру *маршаллинга* (*marshalling*)) и отправляет его по сети вызывающей стороне.

Кроме того, метод `exportObject()` создает и возвращает еще один, другой прокси для этого объекта, который будет загружен в реестр объектов *RMI*. Этот прокси называется *заглушкой* (*stub*). Заглушка знает порт, который прослушивается скелетоном, а также *IP*-адрес компьютера – сервера. Это означает, что у заглушки есть вся необходимая информация о скелетоне, и она может связаться со скелетоном по сети, когда это будет необходимо. Реестр *RMI* объектов для того, чтобы хранить этот объект-заглушку и по запросу клиентов предоставлять им ее. Эта заглушка (*stub*) должна иметь тип *Greeting*, а не *GServer*. Это связано с тем, что заглушка, созданная для удаленного объекта (в нашем случае объекта *server* типа *GServer*) представляет то, как выглядит объект со стороны клиентов, то есть как объект, реализующий интерфейс *Greeting*, в котором перечислены методы, доступные для удаленного вызова. И этот интерфейс, конечно, имплементирован классом *GServer*.

Последовательность установления связи показана на рисунке:



Особенностью метода `exportObject()` является то, что он может не только загружать файлы классов для заглушки и скелетона во время своего вызова, но и в случае, если файлы классов не найдены, динамически создавать их. До *Java 5.0* было необходимо, используя специальный компилятор `rmic`

(<https://docs.oracle.com/javase/8/docs/technotes/tools/windows/rmic.html>),

входящий с состав *JDK*, вручную сгенерировать файлы классов для заглушки и скелетона. В последующих версиях *Java* этот этап стал необязательным и требуемые для работы служебные классы могут быть сгенерированы автоматически во время такого вызова метода `exportObject()`. Этот метод также имеет другую перегруженную версию:

```
public static RemoteStub exportObject(Remote obj)
                                   throws RemoteException
```

В этом варианте вызова метода не требуется указывать номер порта в качестве аргумента и, при этом, ожидается существование предварительно сгенерированных файлов классов для заглушки и скелетона. Имя класса заглушки определяется путем объединения двоичного имени класса удаленного объекта с суффиксом «_Stub», а скелетона – с суффиксом «_Skel». Компилятор `rmic` генерирует файлы классов, используя такое соглашение об именах. Например, следующая команда создает файл класса `all_in_one.GServer_Stub.class` из файла класса реализации `all_in_one.GServer.class`. (Если нужно, то сразу генерируется и скелетон).

```
rmic all_in_one.GServer
```

Если метод `exportObject()` не может найти подходящий класс-заглушку или не может загрузить класс-заглушку, или возникает проблема при создании экземпляра заглушки, то метод выбрасывает исключение `StubNotFoundException`.

Для того, чтобы понять особенности построения этого класса-заглушки, исходный код может быть восстановлен с использованием любого подходящего *декомпилятора Java*. Пример исходного кода, созданного с помощью *JAD Java-декомпилятора*, показан ниже (вообще-то можно указать опцию `-keep` компилятора `rmic`, что бы компилятор сохранил исходный код для созданных класс – файлов):

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
```

```
package all_in_one;
```

```
import java.lang.reflect.Method;
import java.rmi.RemoteException;
import java.rmi.UnexpectedException;
import java.rmi.server.*;
```

```
// Referenced classes of package all_in_one:
//      Greeting

public final class GServer_Stub extends RemoteStub
    implements Greeting
{

    public GServer_Stub(RemoteRef remoteref)
    {
        super(remoteref);
    }

    static Class _mthclass$(String s)
    {
        try
        {
            return Class.forName(s);
        }
        catch(ClassNotFoundException classnotfoundexception)
        {
            throw new
NoClassDefFoundError(classnotfoundexception.getMessage());
        }
    }

    public String greet(String s)
        throws RemoteException
    {
        try
        {
            Object obj = super.ref.invoke(this, $method_greet_0, new
Object[] {
                s
            }, 0x200f41a1529d0462L);
            return (String)obj;
        }
        catch(RuntimeException runtimeexception)
        {
            throw runtimeexception;
        }
        catch(RemoteException remoteexception)
        {
            throw remoteexception;
        }
    }
}
```

```

    }
    catch(Exception exception)
    {
        throw new UnexpectedException("undeclared checked
exception", exception);
    }
}

private static final long serialVersionUID = 2L;
private static Method $method_greet_0;

static
{
    try
    {
        $method_greet_0 =
(all_in_one.Greeting.class).getMethod("greet", new Class[] {
        java.lang.String.class
    });
    }
    catch(NoSuchMethodException _ex)
    {
        throw new NoSuchMethodError("stub class initialization failed");
    }
}
}

```

В настоящее время рекомендуется использовать предыдущую версию метода `exportObject()`, чтобы избежать явного создания класса заглушки. В этом случае система *Java RMI* генерирует прокси-класс (с именем типа `ProxyO`), функциональность которого соответствует функциональности класса-заглушки, сгенерированного компилятором `rmic`. Затем он динамически создает и устанавливает объект прокси (скелетон) для серверного объекта. Кроме этого, метод `exportObject()` также создает и возвращает другой прокси (заглушку) для этого объекта; впоследствии заглушка будет загружена в реестр объектов *RMI*. Приведем исходный код такого автоматически сгенерированного класса – заглушки (соответствующий класс файл можно получить с помощью программы *Java*-агента, а затем восстановить эквивалентный исходный код с помощью декомпилятора):

```

// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)

```

```
package com.sun.proxy;

import all_in_one.Greeting;
import java.lang.reflect.*;
import java.rmi.RemoteException;

public final class $Proxy0 extends Proxy
    implements Greeting
{

    public $Proxy0(InvocationHandler invocationhandler)
    {
        super(invocationhandler);
    }

    public final boolean equals(Object obj)
    {
        try
        {
            return ((Boolean)super.h.invoke(this, m1, new Object[] {
                obj
            })).booleanValue();
        }
        catch(Error _ex) { }
        catch(Throwable throwable)
        {
            throw new UndeclaredThrowableException(throwable);
        }
    }

    public final String toString()
    {
        try
        {
            return (String)super.h.invoke(this, m2, null);
        }
        catch(Error _ex) { }
        catch(Throwable throwable)
        {
            throw new UndeclaredThrowableException(throwable);
        }
    }
}
```

```
public final String greet(String s)
    throws RemoteException
{
    try
    {
        return (String)super.h.invoke(this, m3, new Object[] {
            s
        });
    }
    catch(Error _ex) { }
    catch(Throwable throwable)
    {
        throw new UndeclaredThrowableException(throwable);
    }
}

public final int hashCode()
{
    try
    {
        return ((Integer)super.h.invoke(this, m0, null)).intValue();
    }
    catch(Error _ex) { }
    catch(Throwable throwable)
    {
        throw new UndeclaredThrowableException(throwable);
    }
}

private static Method m1;
private static Method m2;
private static Method m3;
private static Method m0;

static
{
    try
    {
        m1 = Class.forName("java.lang.Object").getMethod("equals",
new Class[] {
        Class.forName("java.lang.Object")
    });
    }
}
```

```

        m2 = Class.forName("java.lang.Object").getMethod("toString",
new Class[0]);
        m3 = Class.forName("all_in_one.Greeting").getMethod("greet",
new Class[] {
            Class.forName("java.lang.String")
        });
        m0 = Class.forName("java.lang.Object").getMethod("hashCode",
new Class[0]);
    }
    catch(NoSuchMethodException nosuchmethodexception)
    {
        throw new
NoSuchMethodError(nosuchmethodexception.getMessage());
    }
    catch(ClassNotFoundException classnotfoundexception)
    {
        throw new
NoClassDefFoundError(classnotfoundexception.getMessage());
    }
}
}

```

Система *Java RMI* генерирует этот *прокси*-класс, используя концепцию *динамического прокси*. Можно отметить, что реализация метода `greet()` в обоих случаях аналогична.

Регистрация заглушки

Прокси объект, размещенный на стороне сервера (скелетон), теперь готов принять входящий запрос на вызов удаленного метода. При этом, была сгенерирована заглушка (*стаб*) удаленного объекта. Эта заглушка может при помощи сокета удаленно взаимодействовать со скелетоном. Таким образом, если бы можно было каким-либо образом получить экземпляр этой заглушки на удаленном компьютере, то можно было выполнить удаленный вызов метода на удаленном объекте с помощью этого экземпляра заглушки. Обычно, заглушка передается клиенту из отдельного приложения, называемого реестром *RMI*, в котором серверная сторона регистрирует заглушку.

Чтобы зарегистрировать заглушку, сгенерированную методом `exportObject()` в *RMI* реестре объектов, необходимо получить ссылку на этот реестр объектов. Можно создать программно новый *RMI* реестр объектов или найти и использовать существующий (уже запущенное внешнее приложение). Класс `java.rmi.registry.LocateRegistry` (см. <https://docs.oracle.com/javase/8/docs/api/java/rmi/registry/LocateRegistry.html>) содержит достаточное количество статических методов для этой цели. В нашем примере воспользуемся существующим реестром объектов, который

можно запустить с помощью приложения `rmiregistry`, которое входит в стандартную поставку *JDK* (вообще-то, и *JRE*). Приложение `rmiregistry` работает как отдельный процесс и позволяет приложениям регистрировать удаленные объекты или получать ссылки на уже зарегистрированные именованные удаленные объекты. Система *Java RMI*, на данный момент, позволяет запускать `rmiregistry` только на том же компьютере, на котором работает *RMI* сервер.

Рассмотрим строку кода, которая используется для получения ссылки на существующий *RMI* реестр объектов, который уже запущен на том же компьютере на порту по умолчанию (1099).

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

Если реестр объектов работает на порте, отличном от 1099, то нужно указать номер порта при вызове метода `getRegistry()`. Например, если *RMI* реестр объектов запущен на заданном порту `port`.

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

Вместо того, чтобы использовать существующий реестр, приложение само может программно создать *RMI* реестр следующим образом:

```
Registry registry = LocateRegistry.createRegistry(1099);
```

Этот фрагмент кода создает *RMI* реестр, который прослушивает порт с номером 1099. Понятно, что реестр, созданный с использованием такого подхода, доступен только при условии, что приложение – сервер уже запущено. Но, в любом случае, когда есть ссылка на реестр, заглушка регистрируется в этом реестре объектов следующим образом:

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

См. справочную статью по интерфейсу `java.rmi.registry.Registry` (<https://docs.oracle.com/javase/8/docs/api/java/rmi/registry/Registry.html>). Есть несколько методов для регистрации, но чаще используется метод `rebind()`, который связывает указанную заглушку и имя и регистрирует указанную заглушку по указанному имени в реестре. Если для указанного имени уже существует привязка, то она просто переопределяется. Таким образом, заглушка для удаленного объекта будет зарегистрирована в реестре под именем «Greet».

Когда клиент запрашивает в реестре объектов ссылку на удаленный объект, то ему передается заглушка для удаленного объекта. Клиентское приложение, которое связалось с реестром объектов, создает экземпляр этой заглушки и устанавливает ее на клиентском компьютере, а затем возвращает

клиенту соответствующую ссылку. Клиент вызывает методы на этом локальном объекте – заглушке. Эта заглушка содержит всю информацию, необходимую для сетевого взаимодействия со скелетоном, который является прокси для удаленного объекта на стороне сервера. Заглушка создает сокет для взаимодействия со скелетоном. Она упаковывает информацию, необходимую для вызова метода, такую как имя метода, его параметры и т.д. (этот процесс называется маршаллингом (*marshalling*)). Затем заглушка отправляет упакованные данные скелетону через созданный сокет. Затем скелет выполняет ан-маршаллинг (*un-marshalling*) данных и выполняет действия, указанные выше.

Создание **RMI** клиента

Клиенты, удаленно взаимодействующие с GServer, относительно просты. По сути, это программа, которая вызывает метод `greet()` для объекта, созданного и экспортированного сервером. Однако, чтобы иметь возможность вызвать метод, удаленный клиент должен сначала получить ссылку на удаленный объект. Ссылка на удаленный объект может быть получена разными способами:

- с помощью реестра;
- с помощью службы именования *RMI (RMI naming service)*;
- за счет передачи и возврата удаленных объектов.

В нашем демонстрационном приложении воспользуемся первым методом. Мы помним, что удаленные ссылки хранятся в реестре объектов. Поэтому, клиент сначала получает удаленную ссылку на реестр объектов, работающий на серверном хосте, с помощью метода `LocateRegistry.getRegistry()` следующим образом:

```
String host = "localhost";
//Registry registry = LocateRegistry.getRegistry(host);
registry = LocateRegistry.getRegistry(host, port);
```

В качестве аргумента метод `getRegistry()` ожидает имя хоста, которое является именем компьютера или *IP*-адресом компьютера, на котором реестр работает на порте по умолчанию (1099). Если реестр работает на порте, отличном от 1099, то в качестве второго аргумента метода `getRegistry()` следует указывать нужный номер порта. Метод `getRegistry()` обращается к удаленному реестру (при помощи сокета) и возвращает локальный прокси для реестра (заглушка для реестра, которая является экземпляром класса `RegistryImpl_Stub`). Следует обратить внимание на то, что этот прокси создается и загружается динамически после загрузки данных класса `RegistryImpl_Stub` из удаленного реестра объектов. Этот прокси-реестр знает всю информацию об удаленном реестре (порт и *IP*-адрес) и может обмениваться данными (с помощью сокета) с удаленным реестром по запросу.

После того, как клиент получит ссылку на удаленный реестр, он может вызвать метод `lookup()` на этой ссылке, чтобы получить ссылку на удаленный объект, зарегистрированный в реестре на стороне сервера следующим образом:

```
String name = "Greet";
Greeting stub = (Greeting) registry.lookup(name);
```

Для получения ссылки на удаленный объект клиент должен использовать то же имя, которое сервер использовал для регистрации объекта. Метод `lookup()`, вызванный на прокси-реестре, обращается к удаленному реестру, загружает заглушку для объекта `GServer` с именем «Greet», указанным в качестве аргумента. Затем он создает и устанавливает экземпляр заглушки (динамически используя концепцию динамической загрузки классов) и возвращает ссылку клиенту. Ссылка `stub` на самом деле является локальной ссылкой на заглушку. Информация об объекте `stub` (например, имя класс – файла) может быть просмотрена с использованием следующего кода:

```
System.out.println(stub);
```

Когда этот оператор будет выполнен, то на экран будет выведено нечто такое:

```
Proxy[Greeting,RemoteObjectInvocationHandler[UnicastRef
[liveRef: [endpoint: [172.16.5.81:52704](remote), objID: [-
1812e356: 1409d599c42: -7fff,1509491269759011786]]]]]
```

Следует обратить внимание, на что `[172.16.5.81:52704]` является адресом сокета скелетона. Эта заглушка находится за сценой и ведет себя как прокси для удаленного серверного объекта. Когда клиент вызывает метод, используя ссылку `stub`, происходит аналогичный вызов метода в заглушке. Затем заглушка создает соединение *TCP*-сокета со скелетоном, маршализует информацию о вызове метода и отправляет запрос. Когда результат вычислений возвращается от скелетона обратно, заглушка возвращает результат клиенту. Вся эта сложная процедура полностью прозрачна для клиента.

Вызов метода `greet()` на удаленном объекте выглядит так же просто, как и вызов метода для обычного локального объекта.

```
String name = "Name";
String answer = stub.greet(name);
```

Приведем полный исходный код клиента, который хранится в файле **GClient.java**.

```

package all_in_one;

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);
            System.out.println("Registry:");
            System.out.println(registry);
            Greeting stub = (Greeting) registry.lookup("Greet");
            System.out.println("Stub");
            System.out.println(stub);
            Scanner in = new Scanner(System.in);
            System.out.print("Your name: ");
            String name = in.nextLine();
            System.out.println(stub.greet(name));
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }
    }
}

```

Запуск распределенного приложения

Мы уже отмечали, что запуск распределенного приложения – это достаточно непростая и громоздкая процедура. Давайте посмотрим, как скомпилировать и запустить это распределенное приложение по частям - от самого простого случая к сложному.

Все классы в одном пакете

Сначала разберемся с процедурой запуска, когда все классы расположены на одном компьютере в одном пакете. Понятно, что в таком случае приложение является «распределенным» просто формально – по используемой технологии. Но именно с такого, самого простого случая, удобно разбирать запуск приложения. Кроме того, все команды, связанные с созданием и запуском распределенного приложения, будут указаны в виде, характерном для операционных систем семейства *Windows*. Для других систем нужно сделать очевидные изменения в синтаксисе команд.

Таким образом, предположим, что текущей директорией является директория `test`, и все исходные файлы расположены в пакете `all_in_one`, который расположен в директории `test`.

Схематически это можно представить так:

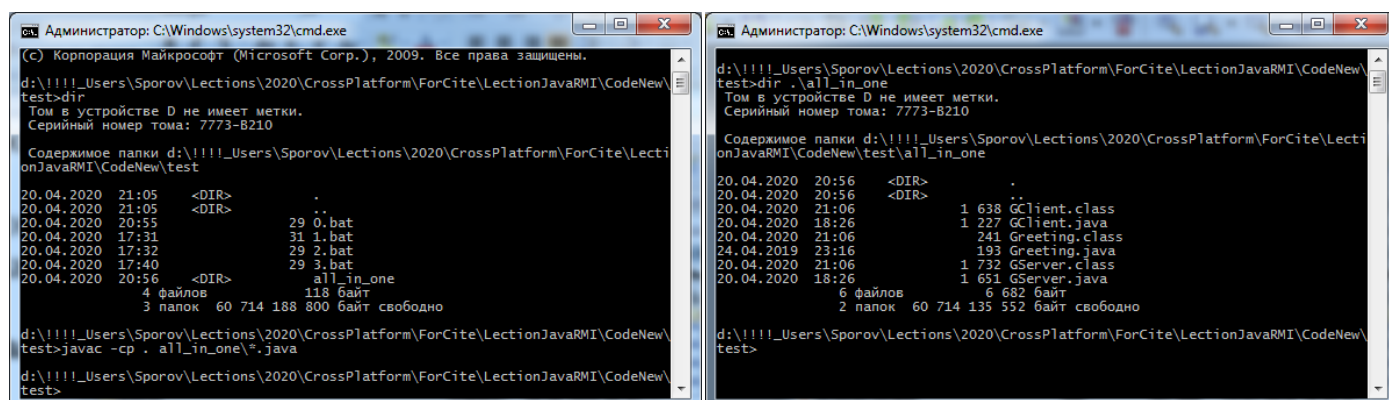
```
test
  all_in_one
    Greeting.java
    GServer.java
    GClient.java
```

Компиляция приложения

Сначала откомпилируем исходные файлы. Находясь в директории `test`, в командном окне введите команду вызова компилятора для обработки исходных файлов *Java*:

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

В директории пакета `all_in_one` будут созданы `*.class` для соответствующий `*.java` файлов с исходным кодом приложения.



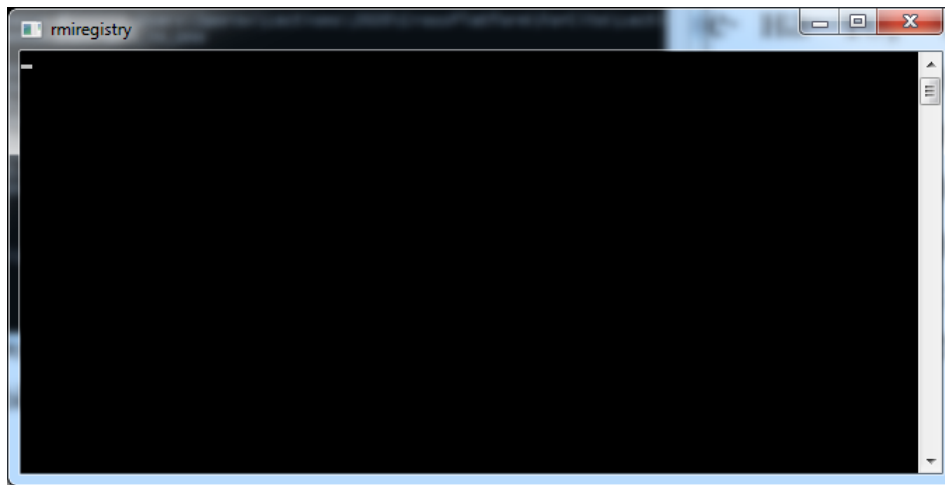
Для удобства работы команду компиляции, да и все остальные команды запуска приложения, можно разместить в текстовых командных файлах (файл с расширением имени `*.bat`).

Запуск службы реестра

Перед запуском собственно приложения, нужно запустить службу *RMI* реестра. В *Java* служба реестра запускается с помощью команды **rmiregistry**. Итак, в терминале, оставаясь на домашнем каталоге (*test*) приложения, введите следующую команду:

```
start "rmiregistry" rmiregistry 6789
```

Данная команда открывает новое окно с заголовком *rmiregistry* и в нем запускает приложение **rmiregistry** на указанном в команде порту (**6789**) (для запуска **rmiregistry** на порту по умолчанию **1099** номер порта вводить не нужно). Если данный порта будет занят, то нужно везде использовать другой, свободный порт. Следует обратить внимание на то, что службе *RMI* реестра **rmiregistry** для работы (регистрации объекта заглушки *Greeting*) нужен файл **Greeting.class**. Поэтому, команду **rmiregistry** следует запускать из каталога, содержащего файл **Greeting.class** (точнее, из каталога, в котором доступен класс *all_in_one.Greeting*). А это именно каталог *test*. Однако, технология *RMI* содержит способы динамической загрузки класса, которые будут подробно представлены позже, при запуске этого приложения с отдельных компьютеров.



Служба *RMI* реестра объектов «молчалива» – по умолчанию, при работе приложения в данном окне не будет выведено ни одного сообщения.

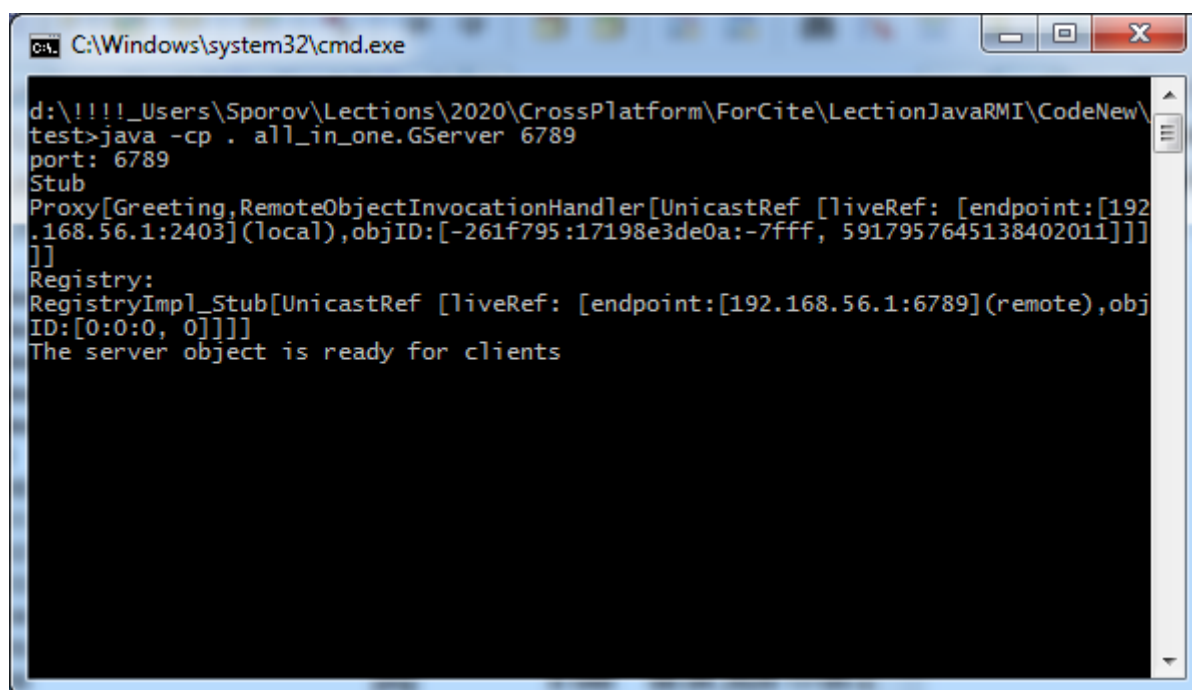
Запуск серверной части приложения

Для того, чтобы запустить сервер нужно в отдельном терминальном окне, находясь в корневой директории нашего приложения **test**, ввести следующую команду:

```
java -cp . all_in_one.GServer 6789
```

Будет запущена серверная часть приложения и выведена информация о порте, ссылке на реестр и заглушку, и текстовая строка, сообщающая о готовности сервера к приему вызовов метода от клиентов.

```
port: 6789
Stub
Proxy[Greeting,RemoteObjectInvocationHandler[UnicastRef
[liveRef: [endpoint:[192
.168.56.1:2403](local),objID:[-261f795:17198e3de0a:-7fff,
5917957645138402011]]]
]]
Registry:
RegistryImpl_Stub[UnicastRef [liveRef:
[endpoint:[192.168.56.1:6789](remote),obj
ID:[0:0:0, 0]]]]
The server object is ready for clients
```



Следует обратить внимание: несмотря на то, что код метода `main` уже выполнен, серверная часть приложения не заканчивает работу. Дело в том, что при экспорте объекта автоматически формируется отдельный поток выполнения (*thread*), в рамках которого работает *RMI* часть технологии. Мы чуть позже разберемся, как можно корректно завершить работу серверной части приложения. А пока будем просто завершать работу терминального окна средствами операционной системы.

Запуск клиентской части приложения

Для запуска клиентской части распределенного приложения запустим терминальное окно и из домашнего каталога приложения (`test`) введем следующую команду:

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

```

C:\Windows\system32\cmd.exe
d:\!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
test>java -cp . all_in_one.GClient localhost 6789
Host: localhost Port: 6789
Registry:
RegistryImpl_Stub[UnicastRef [liveRef: [endpoint:[localhost:6789](remote),objID:
[0:0:0, 0]]]]
Stub
Proxy[Greeting,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:[192
.168.56.1:2403](remote),objID:[-261f795:17198e3de0a:-7fff, 5917957645138402011]]
]]]
Your name: _

```

В результате будет запущен клиент, на экран будет выведена приведенная выше информация и пользователь может ввести имя.

```

Your name: Vasya
Hello, dear Vasya. You are the 1

```

Сделующий запуск приложения ожидаемо отработает.

```

Your name: Petya
Hello, dear Petya. You are the 2

```

Классы в разных пакетах (возможно на разных хостах)

Теперь предположим, что наше удаленное приложение разделено на серверную и клиентскую части, которые размещены в разных папках компьютера (может даже и на разных хостах):

```

some_folder
    different
        server
            all_in_one
                Greeting.java
                GServer.java
            test
        client
            all_in_one
                Greeting.java
                GClient.java

```

Более подробно о структуре каталогов поговорим позже. С помощью такой структуры каталогов мы сделаем следующий шаг для моделирования

распределенного приложения, которое будет работать на нескольких разных компьютерах.

Компиляция

Как и в прошлом примере, сначала давайте откомпилируем файлы с исходным кодом. Этот процесс аналогичен предыдущему этапу – нужно просто перейти в родительский каталог для каталога пакетов (папки **server** / **client** для каталога пакета **all_in_one**):

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

В каталоге пакета **all_in_one** на серверной и клиентской сторонах должны появиться соответствующие ***.class** файлы.

Запуск программы «в старом стиле»

Перед новым запуском *RMI* приложения убедитесь, что все части распределенного приложения, которые были запущены, уже завершили работу. Закройте все открытые, относящиеся к *RMI* приложению, терминальные окна.

На первом этапе запуска распределенного приложения нужно запустить службу *RMI* реестра. Дадим команду на запуск приложения из **rmiregistry** из папки **server** (та папка, в которой находится каталог **all_in_one** с классами, относящимися к серверной части приложения). При таком варианте запуска интерфейс **Greeting**, необходимый службе реестра при регистрации сервером заглушки, будет доступен программе **rmiregistry**. Команды для запуска реестра, для старта серверной и клиентской частей приложения, аналогичны командам, рассмотренным в предыдущем примере. Единственное отличие – местоположение каталога запуска. Теперь это различные каталоги для запуска серверной и клиентской частей приложения. Работа программы и выводимые на экран окна полностью совпадают с предыдущим случаем. Запустите программу в этом случае самостоятельно.

Запуск программы «в новом стиле»

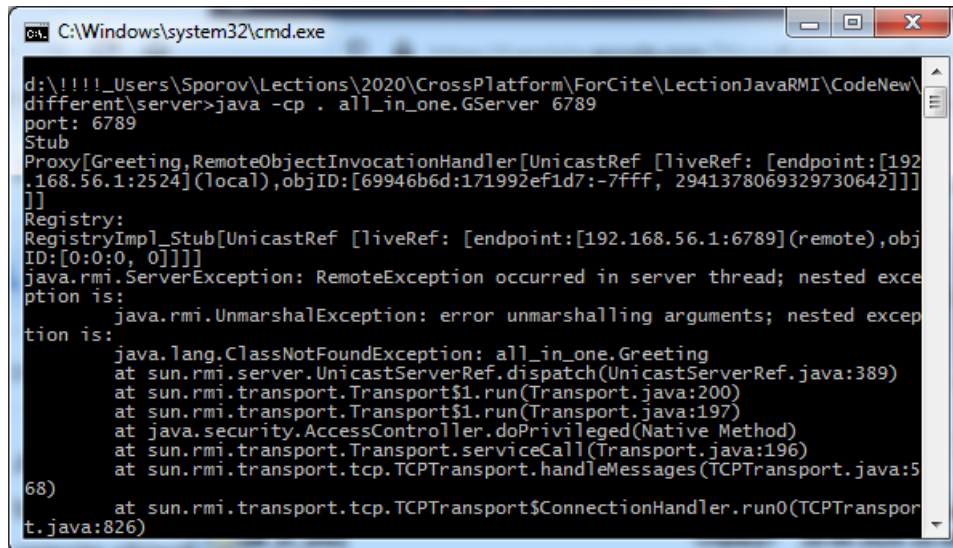
Рассмотрим вариант запуска приложения, когда у службы *RMI* реестра изначально нет доступа к класс-файлу удаленного интерфейса.

Запуск службы реестра

Как и в предыдущем примере, перед запуском серверного приложения нам нужно запустить реестр объектов. Приложение **rmiregistry** может обслуживать удаленные объекты, которые принадлежат различным удаленным приложениям. Для этого, приложению **rmiregistry** нужны класс – файлы удаленных интерфейсов, причем тогда, когда регистрируются соответствующие удаленные объекты. Практически невозможно предусмотреть, какие класс – файлы понадобятся службе реестра в будущем.

Для моделирования такой ситуации, запустим приложение **rmiregistry** из папки, которая не содержит необходимые файлы классов (например, **test**).

Служба реестра будет нормально запущена, но если после этого запустить серверную часть приложения аналогично тому, как это было сделано в предыдущем примере, то будет выброшено большое количество исключений. Дело в том, что службе реестра не нашла класс – файл интерфейса **all_in_one.Greeting**. См. рисунок ниже – исключение **ClassNotFoundException**.



```

C:\Windows\system32\cmd.exe
d:\!!!!_Users\Sporov\Lections\2020\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\
different\server>java -cp . all_in_one.GServer 6789
port: 6789
Stub
Proxy[Greeting,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:[192
.168.56.1:2524](local),objID:[69946b6d:171992ef1d7:-7fff, 2941378069329730642]]]
]]
Registry:
RegistryImpl_Stub[UnicastRef [liveRef: [endpoint:[192.168.56.1:6789](remote),obj
ID:[0:0:0, 0]]]]
java.rmi.ServerException: RemoteException occurred in server thread; nested exce
ption is:
    java.rmi.UnmarshalException: error unmarshalling arguments; nested excep
tion is:
        java.lang.ClassNotFoundException: all_in_one.Greeting
            at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:389)
            at sun.rmi.transport.Transport$1.run(Transport.java:200)
            at sun.rmi.transport.Transport$1.run(Transport.java:197)
            at java.security.AccessController.doPrivileged(Native Method)
            at sun.rmi.transport.Transport.serviceCall(Transport.java:196)
            at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:5
68)
            at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run0(TCPTranspor
t.java:826)

```

Для того, чтобы можно было корректно запустить распределенное приложение нужно сделать доступными требуемые *class* – файлы службе *RMI* реестра. Чтобы передать файлы классов приложению **rmiregistry**, следует использовать свойство **codebase**:

java.rmi.server.codebase

Это свойство указывает месторасположение, из которого могут быть загружены файлы классы, публикуемые этой виртуальной машиной (например, классы-заглушки, пользовательские классы, которые реализуют объявленный возвращаемый тип для вызова удаленного метода, или интерфейсы, используемые прокси-классом или классом-заглушкой). Значением этого свойства является строка в формате *URL*-адреса или разделенный пробелами список *URL*-адресов, который будет кодовой базой для всех классов, загружаемых или локально или маршаллизованных.

Это свойство обязательно должно быть правильно установлено, чтобы динамически загружать классы и интерфейсы при вызове удаленных методов *Java* (*Java RMI*). Если это свойство установлено неправильно, то, скорее всего, будут выброшены исключения при попытке запустить или сервер, или клиент. Для получения дополнительной информации об этом свойстве см. [Dynamic code downloading using Java RMI](https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/codebase.html) (Использование свойства `java.rmi.server.codebase`, <https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/codebase.html>).

Проведем несколько примеров:

1. Если загружаемые классы находятся на *HTTP*-сервере с именем «*webvector*», в каталоге «*export*» (расположенном в корневом *web*-каталоге), то значение свойства **codebase** может быть таким:

```
-Djava.rmi.server.codebase=http://webvector/export/
```

2. Если загружаемые классы находится на *HTTP*-сервере с именем "*webline*", в *JAR* файле с именем "*mystuff.jar*", в каталоге "*public*" (в *web root* сервера), то значение свойства **codebase** может быть таким:

```
-Djava.rmi.server.codebase=http://webline/public/mystuff.jar
```

3. Если же загружаемые классы расположены в двух *JAR* файлах, «*myStuff.jar*» и «*myOtherStuff.jar*», а сами эти *JAR* файлы расположены на разных серверах (с именами «*webfront*» и «*webwave*»), то значение свойства **codebase** может быть таким:

```
-Djava.rmi.server.codebase="http://webfront/myStuff.jar  
http://webwave/myOtherStuff.jar"
```

4. Если же приложение (как клиент, как и сервер) расположено на одном компьютере, можно использовать значение **file:description**:

```
-Djava.rmi.server.codebase=file:/c:/home/ann/classes/
```

Следует учесть, что если не указано иное, то любой вывод из этих свойств отправляется в *System.err*.

Но, в современных версиях *Java* обычно простого указания свойства **java.rmi.server.codebase** недостаточно. При запуске приложения будет возникать впечатление, что *RMI* реестр просто игнорирует свойство *java.rmi.server.codebase*. Дело в том, что, начиная с *JDK 7u21*, свойство *java.rmi.server.useCodebaseOnly* по умолчанию получило значение *true*, тогда как в предыдущих выпусках оно по умолчанию было равно *false*.

Когда свойству *useCodebaseOnly* присвоено значение *false*, то *RMI* реестр (а также и *RMI* клиенты) используют кодовую базу, которая была им передана с сервера. Когда же значение этого свойства равно *true*, то и реестр, и клиенты игнорируют свойство кодовой базы сервера. Реестр и клиенты должны либо сами установить свое собственное значение для *java.rmi.server.codebase*, совпадающее со значением этого свойства у сервера, либо (что не рекомендуется) они могут установить для свойства *useCodebaseOnly* значение *false*. Для получения более подробной информации см. *RMI Enhancements in JDK 7* <https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/enhancements-7.html>.

К сожалению, *RMI Tutorial* на сайте *Oracle* не был обновлен, чтобы отразить эти изменения.

Особенности запуска *RMI* приложений в современной *Java*

Начиная с *JDK 7*, *RMI* свойству `java.rmi.server.useCodebaseOnly` по умолчанию установлено значение `true`. В более ранних выпусках комплекта разработчика значением по умолчанию этого свойства было `false`.

Если этому свойству присвоено значение `false`, то это в этом случае одной стороне *RMI* соединения разрешается указывать местоположение в сети (*URL*), из которого другая сторона *RMI* соединения сможет загрузить *Java* классы. Обычно, механизма динамической загрузки классов используется таким образом, для того, чтобы или *RMI* клиенты, или *RMI* серверы смогли предоставить друг другу удаленные интерфейсы и классы данных без необходимости явной настройки другой стороны взаимодействия.

В этом случае, если *JVM* на одном конце *RMI* соединения указывает один или несколько *URL*-адресов в качестве значения своего системного свойства `java.rmi.server.codebase`, эта информация передается через *RMI* соединение на другой конец взаимодействия. Если для *JVM* на принимающей стороне системное свойство `java.rmi.server.useCodebaseOnly` установлено в значение `false`, то она будет пытаться использовать эти *URL*-адреса для загрузки *Java* классов, на которые имеются ссылки в *RMI* запросах.

Однако, такое поведение – автоматическая загрузка классов из местоположений, указанных на удаленном конце *RMI*-соединения, отключается, если для свойства `java.rmi.server.useCodebaseOnly` установлено значение `true`. В этом случае, классы загружаются только из предварительно настроенных расположений, таких как значение локального свойства `java.rmi.server.codebase` или локального `CLASSPATH`, а не из кодовой базы, передаваемой через *RMI* запросы.

Такое изменение значения свойства по умолчанию может привести к неожиданному сбою приложений на основе *RMI*. Типичным показателем такой проблемы является трассировка стека, содержащая исключение `java.rmi.UnmarshalException`, содержащее вложенное исключение `java.lang.ClassNotFoundException`.

Если при старте *RMI* приложения выбрасываются такие исключения, то следует настроить параметр, определяющий кодовую базу всех клиентов и серверов *RMI*, указав правильные значения в системном свойстве `java.rmi.server.codebase`. Это можно сделать при старте частей приложения с помощью опции `-D` команды, запускающей приложение:

```
java -Djava.rmi.server.codebase=file:///<path-to-remote-classes>/
```

Кроме того, может потребоваться настроить разрешения в файле политики безопасности приложения (*security policy file*), чтобы разрешить доступ к местоположению, указанному в *URL*-адресе. Для этого, обачно, нужно

настроить разрешения (*granting permissions*), для `FilePermission` и `SocketPermission`.

Для того, чтобы настроить службу реестра `rmiregistry` на использование заданной кодовой базы (*codebase*), можно использовать следующий синтаксис:

```
rmiregistry -J-Djava.rmi.server.codebase=file:///<path-to-remote-classes>/
```

Немного другой синтаксис используется при указании кодовой базы для группы активации *JVM (activation group JVMs)*, запущенных посредством демона `rmid`. Сам этот демон не обрабатывает *RMI* запросы, но он создает подпроцессы *JVM* для обработки *RMI* запросов. Синтаксис для указания кодовой базы для подпроцессов `rmid` такой:

```
rmid -C-Djava.rmi.server.codebase=file:///<path-to-remote-classes>/
```

В некоторых случаях может быть или сложно, или даже невозможно предварительно сконфигурировать *RMI* клиенты или *RMI* серверы, а также демоны `rmiregistry` или `rmid` для использования с определенной кодовой базой. В этом случае для устранения таких проблем можно просто установить для свойства `java.rmi.server.useCodebaseOnly` значение `false`. Это можно сделать с помощью таких параметров командной строки:

```
java -Djava.rmi.server.useCodebaseOnly=false
```

Аналогичный синтаксис используется при работе с командами `rmiregistry` и `rmid`.

Но, вообще то, не рекомендуется запускать системы со свойством `java.rmi.server.useCodebaseOnly`, для которого установлено значение `false`, так как это может позволить загрузку и выполнение ненадежного кода (*untrusted code*).

Варианты запуска *RMI* приложения

Способ 1¹

Перед новым запуском *RMI* приложения убедитесь, что все части распределенного приложения, которые были запущены, уже завершили работу. Закройте все открытые, относящиеся к *RMI* приложению, терминальные окна.

Рассмотрим возможный способ запуска распределенного приложения в случае, когда и серверная, и клиентская части приложения расположены на

¹ Данный способ без проблем отработал на Java 8. На более свежих версиях я его не проверял. Если будут проблемы – не страшно. Тогда сообщите об этом и просто внимательно прочитайте. Основной способ запуска *RMI* приложений – следующий. Его разберите подробно и попробуйте, а он точно рабочий.

одном компьютере. В этом случае, в качестве значения кодовой базы можно указать месторасположение файла в файловой системе *ПК*. Лучше указывать «абсолютный путь» - начиная с логического диска и т.д. В этом случае одно и то же значение можно указать для кодовой базы и для службы реестра, и для серверной части приложения. (Для случая *Java 8* этот способ отработал без ошибок).

Выполним запуск службы *RMI* реестра из каталога *test*. В этом случае реестр без указания параметра *codebase* не будет иметь доступ к класс-файлу удаленного интерфейса (см. предыдущий случай запуска). Для этого укажем команду:

```
start "rmiregistry" rmiregistry 6789
-J-Djava.rmi.server.codebase=
"file:/d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForCit
e\LectureJavaRMI\CodeNew\different\server/"
```

Понятно, что путь к каталогу с пакетом *all_in_one*, на каждом компьютере должен быть свой. Для удобства повторного использования эту команду лучше поместить в пакетный файл (*.bat). Будет запущена служба реестра, которая как обычно работает «молчаливо».

После запуска службы реестра запускаем серверную часть. Для этого, из каталога *server* в терминальном окне даем команду на запуск серверной части с тем же значением кодовой базы:

```
java -cp . -Djava.rmi.server.codebase=
"file:///d:\!!!!_Users\Sporov\Lectons\2020\CrossPlatform\ForC
ite\LectureJavaRMI\CodeNew\different\server/"
all_in_one.GServer 6789
```

Будет создан удаленный объект, экспортирован и полученная заглушка разеистрирована в *RMI* реестре. Серверный объект будет доступен клиентам и готов к вызову на нем удаленных методов.

После успешного старта серверной части приложения, запускаем клиентскую часть. Для этого перейдем в каталог *client* и в терминальном окне вводим команду запуска клиента:

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

Будет найдена служба реестра, получена удаленная ссылка и на ней будет вызван удаленный метод.

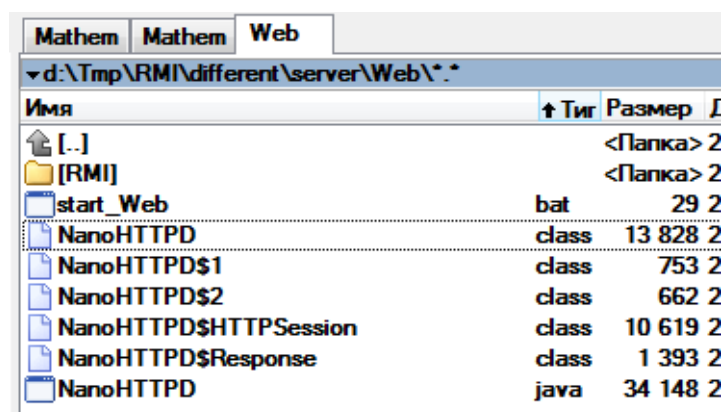
Основные экраны, демонстрирующие работу приложения такие же, как и в прошлом случае.

Способ 2

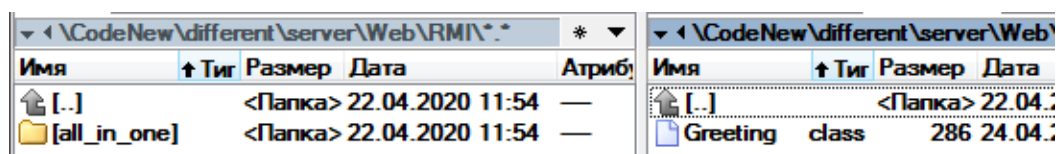
Рассмотрим случай, когда серверная и клиентская части распределенного приложения запущена на разных компьютерах. Вообще-то

именно так рекомендуется запускать *RMI* приложение и в том случае, когда его части расположены на одном компьютере.

Для этого способа запуска понадобится *Web* – сервер, который будет предоставлять необходимые для работы класс – файлы заинтересованной в них стороне. Можно использовать любой *HTTP*-сервер. Для упрощения работы воспользуемся экстремально простым *HTTP*-сервером, полностью реализованном на *Java*: сервером *NanoHTTPD*. Данный сервер реализован в исходном файле **NanoHTTPD.java**, который будет размещен на нашем сайте в разделе практических заданий. Скачайте его в какую-либо папку (у меня **Web**). Вызовите компилятор *Java javac* для компиляции приложения. После компиляции в той же папке появится 5 класс-файлов:



В этой же папке (корневом каталоге сервера) создадим новый каталог *RMI*, куда запишем каталог пакета *all_in_one*, куда разместим только класс-файл удаленного интерфейса **Greeting.class**.

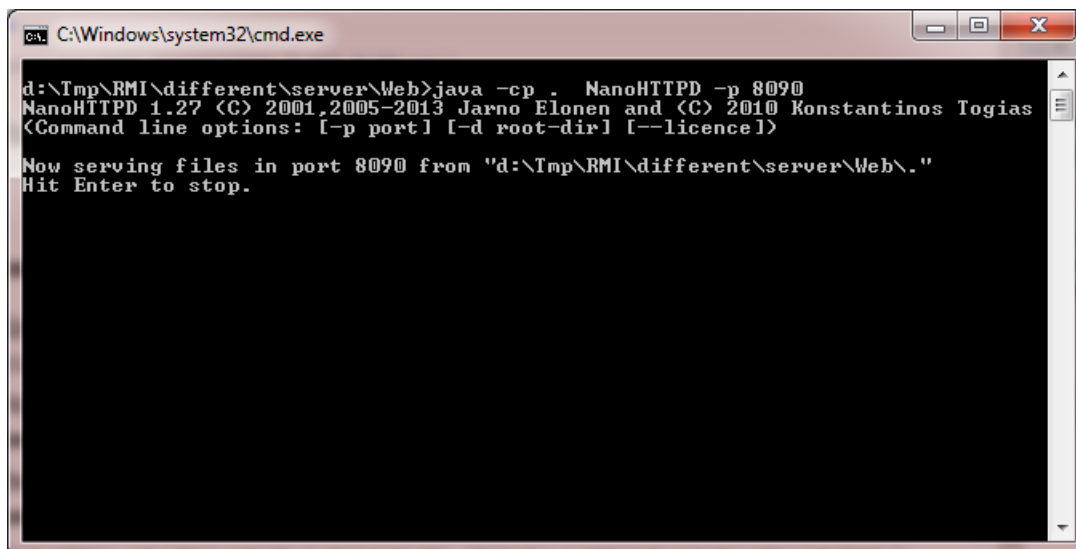


Из каталога **Web** нужно дать команду на запуск сервера. Для удобства эту команду, как и все другие команды, разместите в пакетном файле. На рисунке это командный файл **start_Web.bat** в папке **Web**. В этом файле записана команда запуска данного *HTTP*-сервера:

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

Данная команда запустит *HTTP*-сервер на заданном порту (8090). Понятно, что данный порт должен быть свободным. Если он занят – укажите другой номер – номер свободного порта.

На экран будет выведено информационное окно работы сервера с его основными характеристиками и сообщением, что для завершения работы сервера достаточно нажать на клавишу **<Enter>** в этом окне.



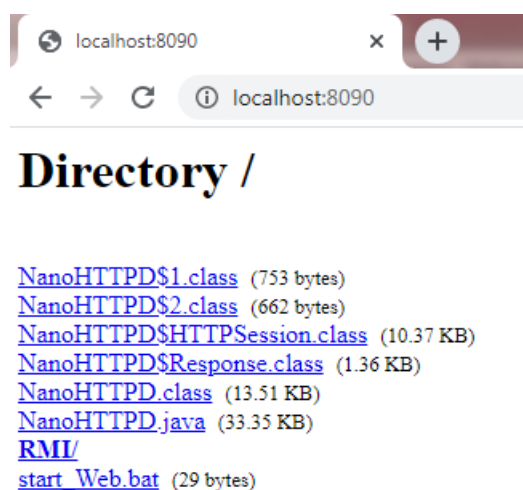
```

C:\Windows\system32\cmd.exe
d:\Tmp\RMI\different\server\Web>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:\Tmp\RMI\different\server\Web\."
Hit Enter to stop.

```

Для проверки работы можно запустить любой браузер и в строке адреса ввести <http://localhost:8090/> . Вы попадете в корневой каталог сервера:



При этом в информационном окне сервера будет выведена информация по запросу:


```

C:\Windows\system32\cmd.exe
HDR: 'upgrade-insecure-requests' = '1'
HDR: 'connection' = 'keep-alive'
HDR: 'accept-encoding' = 'gzip, deflate, br'
HDR: 'accept' = 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9'
HDR: 'sec-fetch-user' = '?1'
HDR: 'sec-fetch-mode' = 'navigate'
HDR: 'accept-language' = 'ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7'
HDR: 'user-agent' = 'Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.113 Safari/537.36'
HDR: 'sec-fetch-dest' = 'document'
HDR: 'host' = 'localhost:8090'
GET '/favicon.ico'
HDR: 'sec-fetch-site' = 'same-origin'
HDR: 'connection' = 'keep-alive'
HDR: 'accept-encoding' = 'gzip, deflate, br'
HDR: 'accept' = 'image/webp,image/apng,image/*,*/*;q=0.8'
HDR: 'referrer' = 'http://localhost:8090/'
HDR: 'sec-fetch-mode' = 'no-cors'
HDR: 'accept-language' = 'ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7'
HDR: 'user-agent' = 'Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.113 Safari/537.36'
HDR: 'sec-fetch-dest' = 'empty'
HDR: 'host' = 'localhost:8090'

```

После того, как сервер запущен, запускаем службу *RMI* реестра. Для этого из каталога **test** запускаем службе реестра:

```

start "rmiregistry" rmiregistry 6789
-J-Djava.rmi.server.codebase="http://localhost:8090/RMI/"

```

Как обычно, в отдельном окне будет запущена «молчаливая» служба реестра.



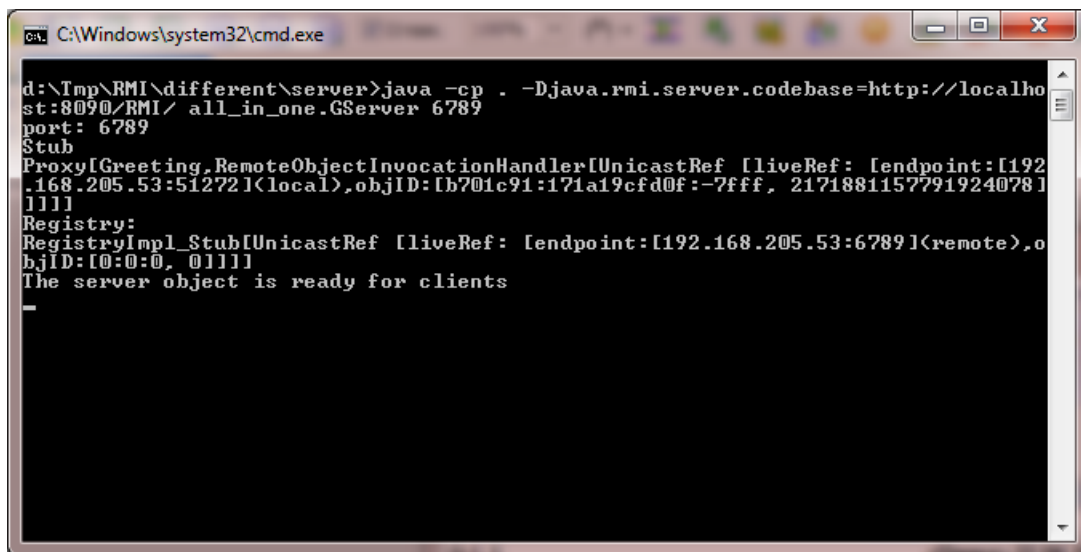
После этого запускаем серверную часть приложения. Для этого из каталога **server** запускаем серверную часть командой:

```

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

```


Серверная часть будет запущена, выполнены все характерные для нее задания. В результате в службе *RMI* реестра будет зарегистрирована заглушка удаленного объекта.

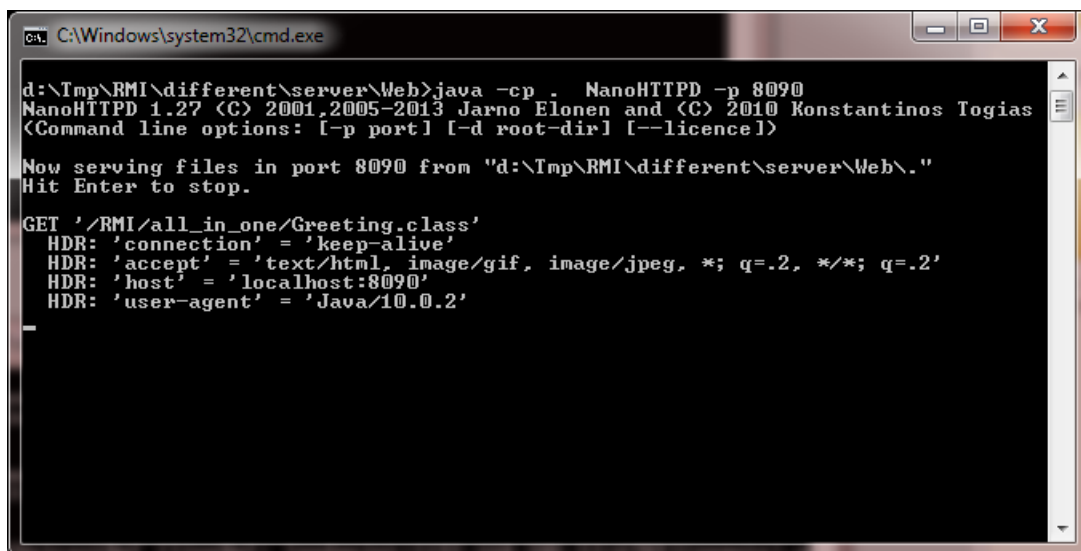


```

C:\Windows\system32\cmd.exe
d:\Tmp\RMI\different\server>java -cp . -Djava.rmi.server.codebase=http://localhost:8090/RMI/ all_in_one.GServer 6789
port: 6789
Stub
Proxy[Greeting,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:[192.168.205.53:51272](local),objID:[b701c91:171a19cfd0f:-7fff, 2171881157791924078]]]]
Registry:
RegistryImpl_Stub[UnicastRef [liveRef: [endpoint:[192.168.205.53:6789](remote),objID:[0:0:0, 0]]]]
The server object is ready for clients

```

При этом, службе *RMI* реестра потребуется класс-файл удаленного интерфейса, который будет получен средствами *HTTP*-сервера, который указан в качестве кодовой базы. Это подтверждается лог-информацией, выведенной в окне сервера:



```

C:\Windows\system32\cmd.exe
d:\Tmp\RMI\different\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:\Tmp\RMI\different\server\Web\."
Hit Enter to stop.

GET '/RMI/all_in_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/10.0.2'

```

После того, как серверная часть будет успешно запущена можно перейти к запуску клиентской части удаленного приложения. Для этого из папки **client** в терминальном окне следует выполнить команду (ее, как и все остальные команды рекомендуется разместить в соответствующем пакетном файле):

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

На экран будет выведено окно клиента:

```

C:\Windows\system32\cmd.exe
d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\different\client
>java -cp . all_in_one.GClient localhost 6789
Host: localhost Port: 6789
Registry:
RegistryImpl_Stub[UnicastRef [liveRef: [endpoint:[localhost:6789]<remote>],objID:
[0:0:0, 0]]]
Stub
Proxy[Greeting,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:[192
.168.205.53:51272]<remote>],objID:[b701c91:171a19cfd0f:-7fff, 2171881157791924078
]]]]
Your name:

```

Введите требуемое имя, посмотрите на результат и запустите клиента еще раз. Посмотрите, что все работает так, как и предполагалось:

```

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\different\client
>3
d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\different\client
>java -cp . all_in_one.GClient localhost 6789
Host: localhost Port: 6789
Registry:
RegistryImpl_Stub[UnicastRef [liveRef: [endpoint:[localhost:6789]<remote>],objID:
[0:0:0, 0]]]
Stub
Proxy[Greeting,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:[192
.168.205.53:51272]<remote>],objID:[b701c91:171a19cfd0f:-7fff, 2171881157791924078
]]]]
Your name: User
Hello, dear User. You are the 2
d:\Sporov\Lections\CrossPlatform\ForCite\LectonJavaRMI\CodeNew\different\client
>_

```

Кстати, после того, как в служба реестра получила нужный класс-файл он (в рамках этого сеанса работы с реестром) повторно загружен не будет. Для проверки этого завершим работу серверной части (закрыв соответствующее окно), завершим работу *HTTP*-сервера (нажмем **<Enter>** в терминальном информационном окне сервера), а службу *RMI*-реестра оставим работающей. Если запустить серверную часть приложения, то она отработает без проблем. И сможет нормально обрабатывать вызовы удаленных методов от клиентов. И это будет именно новый сеанс работы (нумерация клиентов будет начата заново).

```

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

d:\Tmp\RMI\different\client>3.bat
d:\Tmp\RMI\different\client>java -cp . all_in_one.GClient localhost 6789
Host: localhost Port: 6789
Registry:
RegistryImpl_Stub[UnicastRef [liveRef: [endpoint:[localhost:6789]<remote>],objID:
[0:0:0, 0]]]]
Stub
Proxy[Greeting,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:[192
.168.205.53:51379]<remote>],objID:[249f07de:171a1b1c518:-7fff, 746903943145855649
2]]]]]
Your name: Test User
Hello, dear Test User. You are the 1

d:\Tmp\RMI\different\client>_

```

Способ 3

Рассмотрим немного другой способ запуска распределенного приложения, в котором служба реестра получает кодовую базу от сервера. Для этого окончательно закроем все части запущенного ранее распределенного приложения.

Начало запуска – аналогичное прошлому этапу. Как и раньше из папки **Web** запускаем *HTTP*-сервер.

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

После этого из папки **test** запускаем службу реестра, но уже другой командой:

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

Этим мы говорим, что службе реестра будет получать значение кодовой базы по *RMI* каналу во время обработки *RMI* запросов. Затем запускаем серверную часть при помощи той же команды, что и раньше:

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

Аналогично предыдущему случаю при регистрации заглушки удаленного объекта служба реестра получит нужный класс-файл из кодовой базы – указанного при старте сервера значение параметра **codebase**: *Web*-сервера, о чем будет сообщено в его информационном окне.

Затем, аналогично тому, как это делалось ранее, запускаем клиентскую часть приложения. Переходим в папку **client** и в терминальном окне вводим команду запуска *RMI* клиента:

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

Будет запущен клиент, который, как и раньше, выполнит свою задачу.