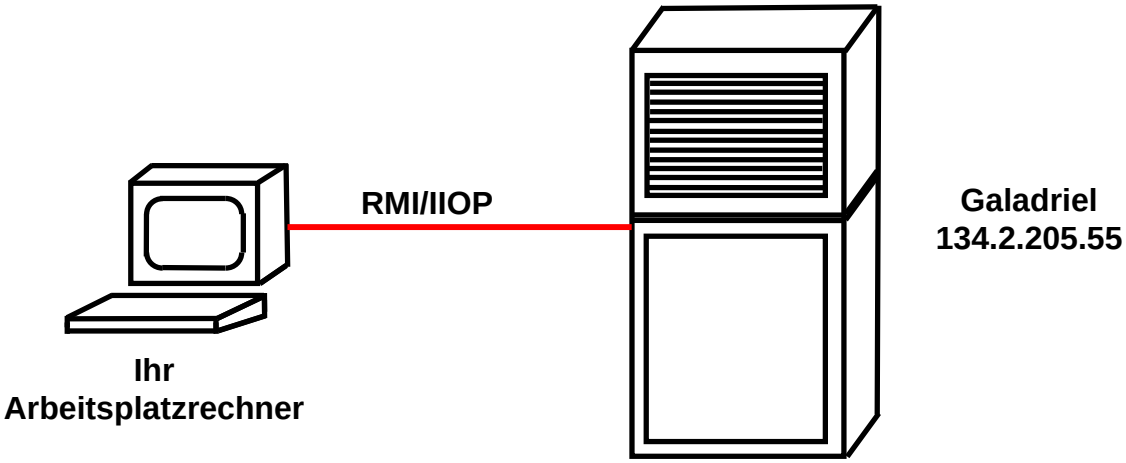
JAVA Remote Method Invocation
RMI/IIOP Tutorial

12.6.2013
Version 0.8

Dieses Tutorial stellt die Aufgabe, ein Java-Programm auf einem Client (Ihrem PC) zu installieren, und mittels RMI einen Zugriff auf einen Java-Server durchzuführen, der in einer zLinux-LPAR auf unserem z9 Mainframe in Tübingen läuft. Die zLinux-LPAR hat den Namen

galadriel.cs.uni-tuebingen.de (oder 134.2.205.55) .

Danksagung an Herrn Robert Harbach für die Bereitstellung des Materials.



Im Gegensatz zu dem JRMP-Tutorial verwenden wir hier RMI/IIOP (gesprochen „RMI over IIOP“). Wir gehen davon aus, dass Sie vor Bearbeitung dieses Tutorials vorher das JRMP-Tutorial bearbeitet haben.

Übersicht

1. Das RMI/IIOP Programmiermodell

1.1 Corba

Der Corba Remote Procedure Call wurde als Nachfolger für die in den 80er Jahren dominierenden Sun RPCs und DCE RPCs geplant und entwickelt. Corba Version 1.0 ist seit 1991 verfügbar, Corba 2.0 seit 1992 und Corba 3.0 seit 2002.

Die wichtigsten Eigenschaften von Corba sind:

- **Die Services eines Corba-Servers werden als Objekte dargestellt und über Objekt-Klassen und deren Methodenaufrufe implementiert.**
- **Corba-Klassen (Binaries) können in vielen Sprachen geschrieben werden (z.B. Cobol, C++, PLI, ADA und Java) und einwandfrei miteinander kommunizieren.**
- **Um dies zu ermöglichen, werden die Schnittstellen der Server-Objekte in einer einheitlichen Schnittstellensprache, der Corba-IDL (Interface Definition Language) beschrieben, unabhängig davon, in welcher Programmiersprache das Server-Objekt geschrieben wurde. Die IDL wird benutzt, um Server-Skeletons und evtl. Client-Stubs zu erstellen.**
- **Auf dem Client und Server wird je eine einheitliche Laufzeitumgebung benötigt, der Corba-ORB (Object Request Broker).**
- **Alle Server-Objekte werden über eine eine einheitliche Bezeichnung adressiert, der IOR (Interoperable Object Reference).**
- **Für die Kommunikation zwischen RMI-Objekten und/oder Corba-Objekten wird das IIOP *(*Internet Inter-Orb Protocol) verwendet.**

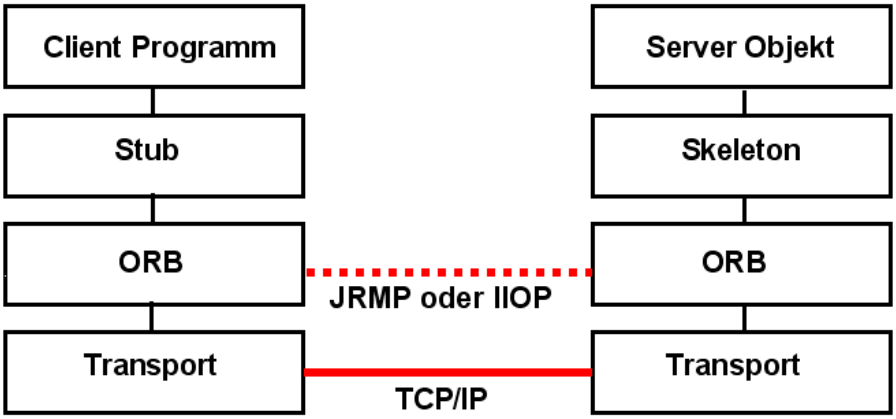
Zusätzlich existieren weitere Corba-Services, z.B. Transaction Service, Naming Service, usw.

IIOP, JRMP, Telnet, 3270, FTP, http und SOAP sind Protokolle der OSI-Layer-Schicht 5 und verwenden heute fast immer TCP/IP als Schicht-4-Übertragungsprotokoll.

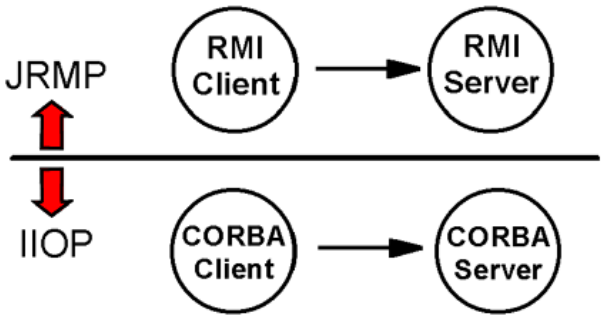
1.2 Java Remote Procedure Call

Bei der Entwicklung der Java-RMI entstand ein mit Corba sehr vergleichbares, aber inkompatibles Programmiermodell. Spezifisch wurde das proprietäre JRMP-Protokoll und eine dazu passende Laufzeitumgebung geschaffen. An Stelle der Corba-IDL benutzt man die Java-Schnittstelle, die Bestandteil von Java ist. Der Vorteil dieser Vorgehensweise ist eine etwas einfachere Programmierung.

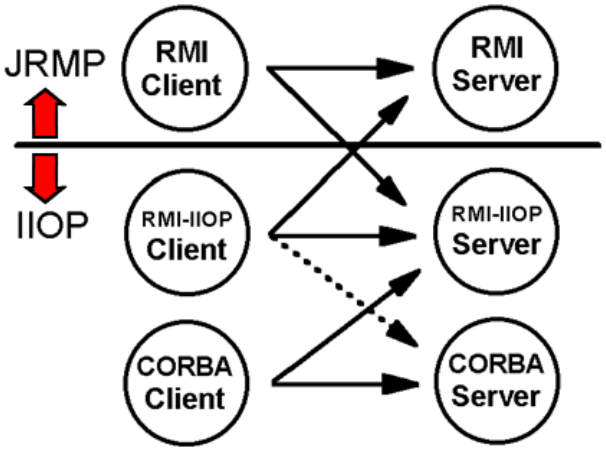
Da es Java RMI over JRMP an Interoperabilität mit anderen Sprachen mangelt, wurde zusätzlich RMI/IIOP geschaffen. Hierzu wurde die JRMP-Laufzeitumgebung durch einen Corba-ORB, und das JRMP-Protokoll durch das IIOP-Protokoll ersetzt. Damit ist es möglich, RMI-Objekte in eine Corba-Umgebung einzubinden. RMI-Objekte benutzen das IIOP-Protokoll, um mit Corba-Objekten zu kommunizieren.



RMI/IIOP bietet den Vorteil, Remote-Interfaces einfach in Java zu schreiben und diese mit den APIs von Java-RMI zu implementieren. Es ist nicht nötig, eine separate IDL-Sprache zu lernen.



JRMP hat einen Nachteil. Ohne RMI/IIOP sind RMI und Corba zwei getrennte Welten. JRMP-Clients können nur mit JRMP-Servern, und Corba-Clients nur mit Corba-Servern kommunizieren.



Die Einführung von RMI/IIOP verbessert diese Situation. Ohne Änderung von existierendem Code ist es möglich, dass JRMP-Cients mit RMI/IIOP-Server-Objekten kommunizieren, und umgekehrt. In anderen Worten, JRMP und RMI/IIOP passen nahtlos zusammen. Weiterhin können Corba-Client-Anwendungen problemlos auf RMI/IIOP-Server-Objekte zugreifen. Beim Zugriff von RMI/IIOP-Clients auf Corba-Server-Objekte existieren einige Einschränkungen, wenn es sich um älteren Code handelt.

Der Java-RMI-Standard offeriert heute zwei alternative Protokolle: JRMP und RMI/IIOP. Manche Softwareunternehmen implementieren beides in ihren RMI-Produkten. IBM hat entschieden, in ihren Java-Produkten ausschließlich RMI/IIOP einzusetzen.

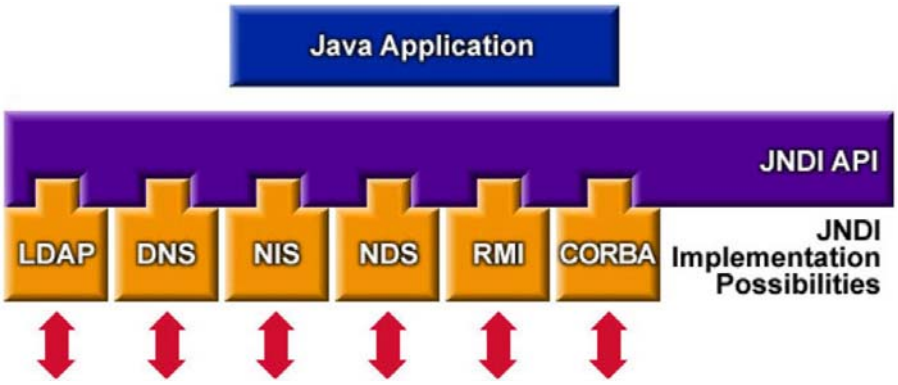
Man sollte meinen, dass der erhöhte Funktionsumfang von RMI/IIOP gegenüber JRMP eine schlechtere Performance zur Folge hat. Dies ist interessanterweise nicht unbedingt der Fall. Die Frage des Unterschieds in der Performance wird im Internet und in den einschlägigen Foren heiß und kontrovers diskutiert und hängt wohl auch von der konkreten Anwendungssituation ab. Es scheint aber so zu sein, dass in der Mehrzahl der Fälle RMI/IIOP in Bezug auf Performance besser abschneidet als JRMP.

Ein ORB (Object Request Broker) ist eine Komponente eines Betriebssystems. Er stellt ein einheitliches Klassenmodell sowie Standards für die Organisation von Klassen-Bibliotheken und die Kommunikation zwischen Objekten zur Verfügung. Alle z/OS-Java-Produkte, z.B. CICS-Java oder der WebSphere-EJB-Container enthalten deshalb einen Corba-ORB.

1.3 Corba und RMI

Es existieren einige Unterschiede in der Programmierung von JRMP- und RMI/IIOP-Anwendungen.

1.3.1 Naming



JNDI (Java Naming and Directory Interface) ist eine API, über die ein Java-Programm einen Namens- und Directory-Dienst in Anspruch nehmen kann. Der Namens- und Directory-Dienst ermöglicht einen Lookup von RMI-Objekten zwischen entfernten JVMs. Mögliche Implementierungen der JNDI-Schnittstelle sind z.B. der LDAP-Server, Corba-Common-Object-Services-Name- (COSNaming-) Server oder der rmiregistry-Server. rmiregistry ist ein Bestandteil des JDK. JRMP verwendet standardmäßig einen rmiregistry-Server.

RMI/IIOP benutzt statt dessen einen COSNaming-Server, welcher Remote-References als Interoperable-Object-References (IORs) enthält. Es existieren verschiedene COSNaming-Server. Wir verwenden den weit verbreiteten `tnameserv`, ebenfalls Bestandteil der JDK.

Das Kommando `tnameserv -ORBInitialPort` startet auf Default-Port 900 (wenn nichts anderes spezifiziert ist).

RMI erwartet aber einen RMI-Name-Server. Deshalb benötigen wir eine JNDI-nach-COSNaming-Bridge. Die `com.sun.jndi.cosnaming.CNCtxFactory`-Bridge ist ein Bestandteil der Sun-JDK, die auf Galadriel installiert ist. Andere Java-Produkte verwenden eine andere Bridge, WebSphere z.B. verwendet einen eigenen Name-Server und eine eigene Bridge.

1.3.2 `rmic`

Der rmic-Compiler erhält die zusätzliche Option `–iiop`. Damit werden Corba kompatible Stubs und Skeletons generiert. Weiterhin wird eine Corba-IDL von dem Java-Interface erzeugt.

1.3.3 Unterschiede in der Codierung

Unterschiedliche Import-Statements werden benutzt (vergleichen Sie die Code-Beispiele der JRMP- und RMI/IIOP-Tutorials).

Das Server-Objekt verwendet anstatt der `UnicastRemoteObject`- die `PortableRemoteObject`-Klasse.

1.3.4 Unterschiede in der Programm-Ausführung

Siehe Teil 4 des Tutorials. Vergleichen Sie es mit dem Teil 4 des JRMP-Tutorials.

1.3.5 Literatur

RMI-IIOP home page
http://java.sun.com/products/rmi-iiop/index.html (obsolet)

RMI-IIOP Programmer's Guide
http://docs.oracle.com/javase/7/docs/technotes/guides/rmi-iiop/rmi_iiop_pg.html

OMG Java language to IDL Mapping specification
ftp://ftp.omg.org/pub/docs/ptc/99-03-09.pdf (obsolet)
http://www.omg.org/spec/JAV2I/

2. RMI/IIOP Programmierung

2.1 Übersicht

Modifizieren Sie ihre bestehende RMI-JRMP-Anwendung so, dass sie das IIOP-Protokoll benutzt und damit Interoperabilität mit Corba-ORBs bietet. Sichern Sie am besten Ihre alte RMI/JRMP-Anwendung, bevor Sie sie modifizieren.

RMI/IIOP macht gegenüber dem JRMP-Tutorial die folgende Änderungen notwendig:

- **Ihre KontoImpl-Klasse erweitert anstatt der UnicastRemoteObject- die PortableRemoteObject-Klasse aus dem javax.rmi-Package**
- **Anstatt des rmiregistry-Name-Servers wird diesmal der Corba kompatible COSNaming-Server benutzt:**

```
> tnameserv -ORBInitialPort <port>
```

- **Ihr Server muss in der main-Methode einen Kontext für JNDI erstellen:**

```
import javax.naming.InitialContext;
InitialContext INC = new InitialContext();
```

Das Binden des Remote-Objekts funktioniert mit der rebind-Methode des InitialContext-Objekts und ersetzt die „Registry“ aus der letzten Aufgabe:

```
INC.rebind("Konto", KontoImpl_object);
```

- **Erzeugen Sie auf die gleiche Art einen JNDI-Kontext in der main-Methode des Clients. InitialContext bietet eine entsprechende „lookup“-Methode um eine Referenz von entfernten Objekten zu bekommen. Sie verwenden dann nicht mehr die lookup-Methode der Naming-Klasse.**

- **Nach dem Kompilieren Ihrer Klassen rufen Sie den RMI-Compiler mit der Option -iiop auf:**

```
> rmic -iiop KontoImpl
```

- **Der Server wird wie folgt gestartet:**

```
> java -
Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFa
ctory -Djava.security.policy=<POLICYFILE> -
Djava.naming.provider.url=iiop://<SERVER:PORTNR> KontoImpl
&
```

(Beachten Sie: der Server nimmt keinen Port-Parameter mehr entgegen, ändern Sie also entpsrechend ihre main-Methode)

- **Der Client wird analog zum Server gestartet**

2.2 Ihre Aufgabe: Eine verteilte Anwendung mit RMI/IIOP

2.2.1 Szenario

Wir verwenden das gleiche Szenario wie in dem JRMP-Tutorial.

2.2.2 Code Beispiel

Quellcode-Beispiele für drei Dateien

- **KontoImpl.java,**
- **Terminal.java,**
- **Konto.java,**

sowie eine Security-Policy-Datei sind im Anhang wiedergegeben.

2.2.3 Port Nummer

Server-Anwendungen erhalten Nachrichten über Ports. Das obige Beispiel verwendet Port-Nr. 2012.

Mehrere Studenten werden gleichzeitig dieses Tutorial bearbeiten. Wir benutzen auf Galadriel aber keine Middleware wie z.B. WebSphere. Stattdessen startet jeder Benutzer auf Galadriel seinen eigenen Server. Damit braucht jeder Benutzer für seinen Server eine eigene Port-Nummer.

Wir schlagen vor, dass Sie eine Port-Nr. im Bereich zwischen 50001 und 50999 wählen.

Um Konflikte mit anderen Benutzern zu vermeiden, schlagen wir vor, dass Sie als Port-Nr. 50xxx wählen, wobei xxx die drei letzten Ziffern Ihrer prak-User-ID sind. Wenn Sie also die User-ID prak519 benutzen, wäre Ihre Port Nr. 50519.

Beachten Sie, dass die obigen Beispielprogramme Port Nr. 2012 benutzen. Sie müssen diese, und auch überall sonst in diesem Text, durch Ihre eigene Port Nr. ersetzen.

- **Bitte fahren Sie Vor Abschluss des Tutorials den von Ihnen gestarteten tnameserv-Prozess wieder herunter!**

2.2.4 Vorgehen

2.2.4.1 Benötigter Quellcode

Erstellen Sie ein eigenes leeres Verzeichnis auf Ihrem Rechner, z.B. iiop, und erstellen Sie die 3 java-Dateien

- **Konto.java**
- **KontoImpl.java**
- **Terminal.java**

in diesem Verzeichnis, sowie zusätzlich die Security-Policy-Datei .

Hinweis:

Eine rudimentäre Implementierung, welche obigen Anforderungen genügt, reicht aus. Auf PIN, Login, mehrere Kontos pro Server, Setzen/Prüfen der Kreditlinie, Zinsen usw. kann verzichtet werden.

Wenn Sie wollen, können Sie dies natürlich trotzdem implementieren.

2.2.4.2 Das Interface und die Klassen kompilieren

Sie müssen die Klassen

- **Konto.class**
- **KontoImpl.class**
- **Terminal.class**

mit Hilfe des `javac`-Compilers erzeugen. Hierfür brauchen Sie eine Java-Entwicklungsumgebung. Sie können (für die primitive Aufgabenstellung ausreichend) hierfür die JDK benutzen, oder eine komfortablere Entwicklungsumgebung wie Eclipse einsetzen.

2.2.4.3. Java Kompatibilität

Genauso wie in dem JRMP-Tutorial lösen wir das Problem der inkompatiblen Java-Versionen, indem wir das Kompilieren aller Klassen auf dem Server durchführen, und die Client-Klassen anschließend in die Workstation laden.

2.2.4.4. Stubs und Skeletons mit rmic erstellen.

Mit Hilfe der Schnittstellenbeschreibung ist es möglich, Stubs und Skeletons zu erstellen. Dies geschieht mit Hilfe des `rmic`-Compilers. Das Java-Interface (in diesem Beispiel „Konto“) Ihres Service-Implementation-Programms enthält alle Informationen, die der Compiler benötigt.

Den rmic-Compiler müssen Sie mit mit der `–iiop` Option

```
rmic –iiop KontoImpl
```

aufrufen. Als Ergebnis werden iiop kompatible Stubs und Skeletons erstellt. Es sollten die Dateien

_KontoImpl_Tie.class
_Kontol_Stub.class

zusätzlich erstellt werden. Die so entstandenen 5 class-Dateien

Konto.class
KontoImpl.class
Terminal.class
_KontoImpl_Tie.class
_Kontol_Stub.class

sind das Ergebnis.

Als nächste Schritte müssen diese Dateien jetzt erzeugt und dann ausgeführt werden.

3. Erstellen der Java-Klassen auf dem zLinux-Server

3.1 Laden der Java-Klassen auf den zLinux-Server

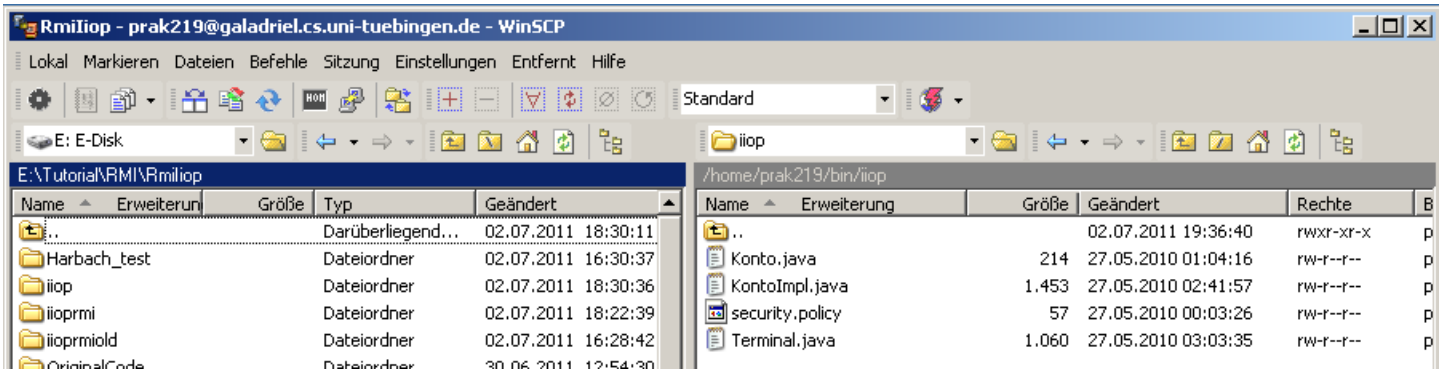
Die 4 Dateien

- **Konto.java**
- **KontoImpl.java**
- **Terminal.java**
- **security.policy**

müssen nun auf dem Server galadriel.cs.uni-tuebingen.de (oder 134.2.205.55) geladen werden. Das Laden geschieht am Einfachsten mit WinSCP.

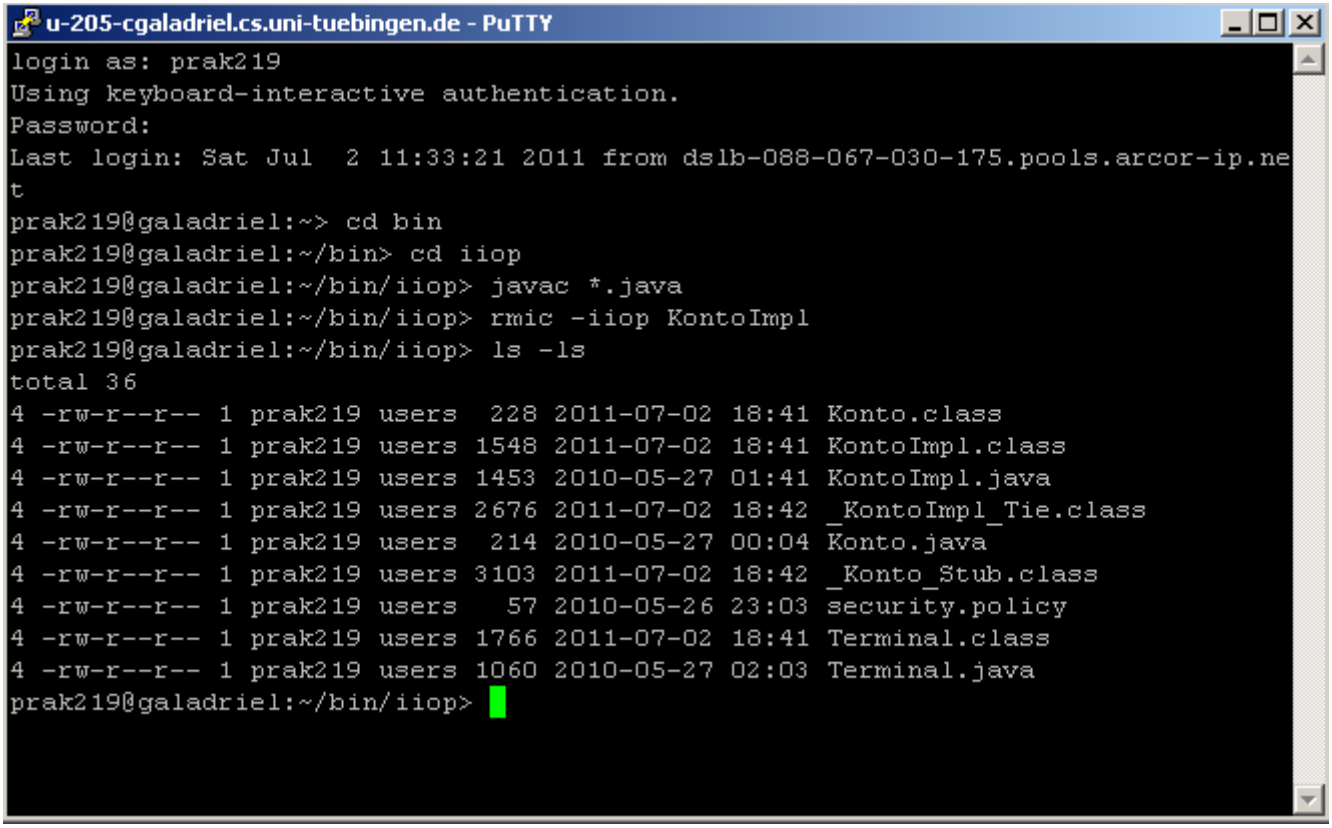
Wir arbeiten auch in diesem Tutorial mit einfachen Java-Klassen, keinen Servlets und EJBs. An Stelle der komplexeren Web-Application-Server-Installation mit jar- und ear-Files genügt ein einfaches Kopieren.

Kopieren Sie wie in dem JRMP-Tutorial das von Ihnen erstellte Verzeichnis iiop nach Galadriel.

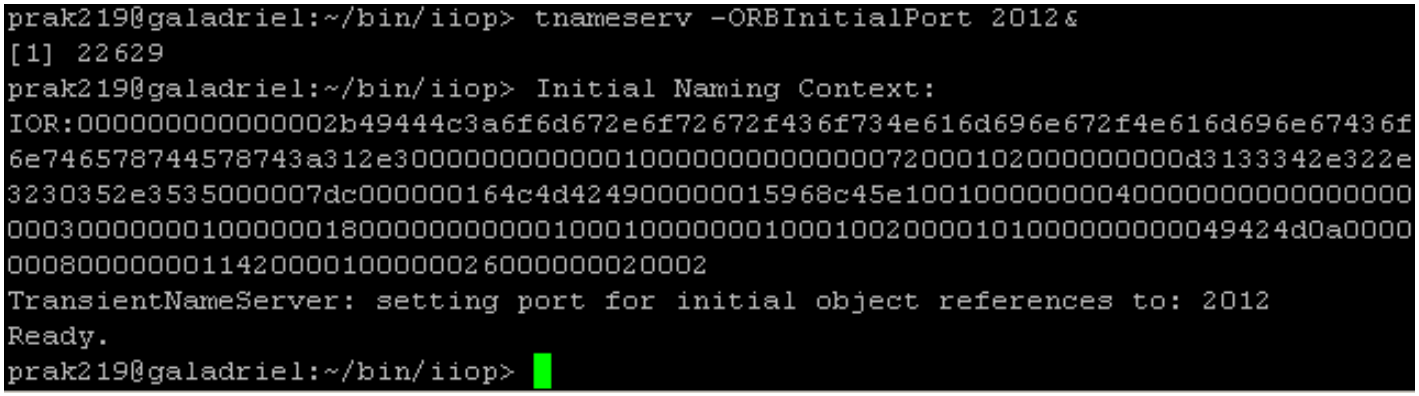


3.2 Zugriff auf zLinux

Als nächstes müssen wir den Quellcode kompilieren. Rufen Sie wieder PuTTY auf und loggen Sie sich ein.



Java-Klassen sowie Stub und Tie werden erstellt.



Wir müssen auch hier definieren, welchen Port wir verwenden wollen. Ähnlich zur Java-Registry brauchen wir einen Eintrag in einen Corba-Name-Service. Sie müssen daher vor dem Aufruf des Servers (KontoImpl) auf Galadriel den Befehl `tnameserv -ORBInitialPort 2012&` eingeben.

`tnameserv -ORBInitialPort 2012&` ist das RMI/IIOP-Pendant zum JRMP-Registry-Aufruf `rmiregistry 2012&` aus dem JRMP-Tutorial.

Der Corba-COS- (Common Object Services) Naming-Service besteht aus einer baumartigen Ordner-Struktur für Object-References, vergleichbar mit der Ordner-Struktur für Dateien in einem Linux-Dateisystem. Der Transient-Naming-Service `tnameserv` ist eine einfache Implementierung der COSNaming-Service-Specifikation. Die Option `-ORBInitialPort` spezifiziert die Portnummer, auf der der Server Nachrichten empfängt.

Das „&“ am Ende des Command-Strings bewirkt wie im JRMP-Tutorial, dass für die Ausführung ein getrennter Hintergrundprozess gestartet wird. Die Ziffer 22629 ist die PID (Prozess-ID.

Der Begriff IOR bedeutet Interoperable Object Reference. Eine IOR beschreibt eine Objektreferenz auf ein Corba-Objekt. Der Corba-Naming-Service ist eine Möglichkeit zur Speicherung von Objekt-Referenzen unter einem Namen. IOR und Name sind vergleichbar mit DNS-IP-Adressen
(z.B. 134.2.205.55) und symbolischen Namen (z.B. galadriel.cs.uni-tuebingen.de). Während aber IP-Adressen 32 Bit lang sind (128 Bit in Version 6), haben IORs typischerweise eine Länge von mehreren 100 Hex-Zeichen. In unserem Fall ist die IOR etwa 300 Zeichen lang.

IP-Adressen werden vom NIC (oder DENIC) zentral verwaltet, während IORs vom Programmierer frei vergeben werden. Die sehr langen IORs schließen die zufällige Erstellung von Synonymen praktisch aus.

Ready bedeutet, dass der Naming-Service läuft.

```
prak219@galadriel:~/bin/iiop> tnameserv -ORBInitialPort 2012&
[1] 22629
prak219@galadriel:~/bin/iiop> Initial Naming Context:
IOR:000000000000002b49444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436f
6e746578744578743a312e3000000000000010000000000000072000102000000000d3133342e322e
3230352e3535000007dc000000164c4d424900000015968c45e100100000000400000000000000000
0003000000010000001800000000000010001000000010001002000010100000000000049424d0a0000
000800000001142000010000002600000000200002
TransientNameServer: setting port for initial object references to: 2012
Ready.
prak219@galadriel:~/bin/iiop>
```

<Enter> bringt uns zurück zum Prompt.

```
prak219@galadriel:~/bin/iiop> tnameserv -ORBInitialPort 2012&
[1] 22629
prak219@galadriel:~/bin/iiop> Initial Naming Context:
IOR:000000000000002b49444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436f
6e746578744578743a312e3000000000000010000000000000072000102000000000d3133342e322e
3230352e3535000007dc000000164c4d424900000015968c45e100100000000400000000000000000
0003000000010000001800000000000010001000000010001002000010100000000000049424d0a0000
000800000001142000010000002600000000200002
TransientNameServer: setting port for initial object references to: 2012
Ready.
prak219@galadriel:~/bin/iiop> java -Djava.naming.factory.initial=com.sun.jndi.co
snaming.CNCtxFactory -Djava.security.policy=security.policy -Djava.naming.provid
er.url=iiop://galadriel.cs.uni-tuebingen.de:2012 KontoImpl
```

Als nächsten Schritt starten wir den KontoImpl-Server mit der Eingabe des folgenden Kommandos:

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
     -Djava.security.policy=security.policy
     -Djava.naming.provider.url=iiop://galadriel.cs.uni-tuebingen.de:2012
     KontoImpl
```

`java KontoImpl` ruft die Klasse `KontoImpl.class` auf. Dies geschieht über drei `–Djava` Qualifier (Optionen). Djava-Qualifier werden benutzt, um Eigenschaften an die aufgerufene JVM zu übergeben. In diesem Fall sind es die Eigenschaften

- **naming.factory.initial**
- **security.policy**
- **naming.provider.url**

denen die hinter dem Gleichheitszeichen stehenden Werte übergeben werden. Bei Verwendung des Sun-JDKs ist `com.sun.jndi.cosnaming.CNCtxFactory` ein gültiger Wert für die `naming.factory.initial` Eigenschaft.

Details zu Properties finden Sie unter http://download.oracle.com/javase/tutorial/essential/environment/properties.html .

```
prak219@galadriel:~/bin/iiop> tnameserv -ORBInitialPort 2012&
[1] 22629
prak219@galadriel:~/bin/iiop> Initial Naming Context:
IOR:000000000000002b49444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436f
6e746578744578743a312e3000000000000010000000000000072000102000000000d3133342e322e
3230352e3535000007dc000000164c4d424900000015968c45e1001000000004000000000000000000
0003000000010000001800000000000010001000000010001002000010100000000000049424d0a0000
0008000000011420000100000026000000020002
TransientNameServer: setting port for initial object references to: 2012
Ready.
prak219@galadriel:~/bin/iiop> java -Djava.naming.factory.initial=com.sun.jndi.co
snaming.CNCtxFactory -Djava.security.policy=security.policy -Djava.naming.provid
er.url=iiop://galadriel.cs.uni-tuebingen.de:2012 KontoImpl
KontoObj bound to registry
KontoServer bereit.
```

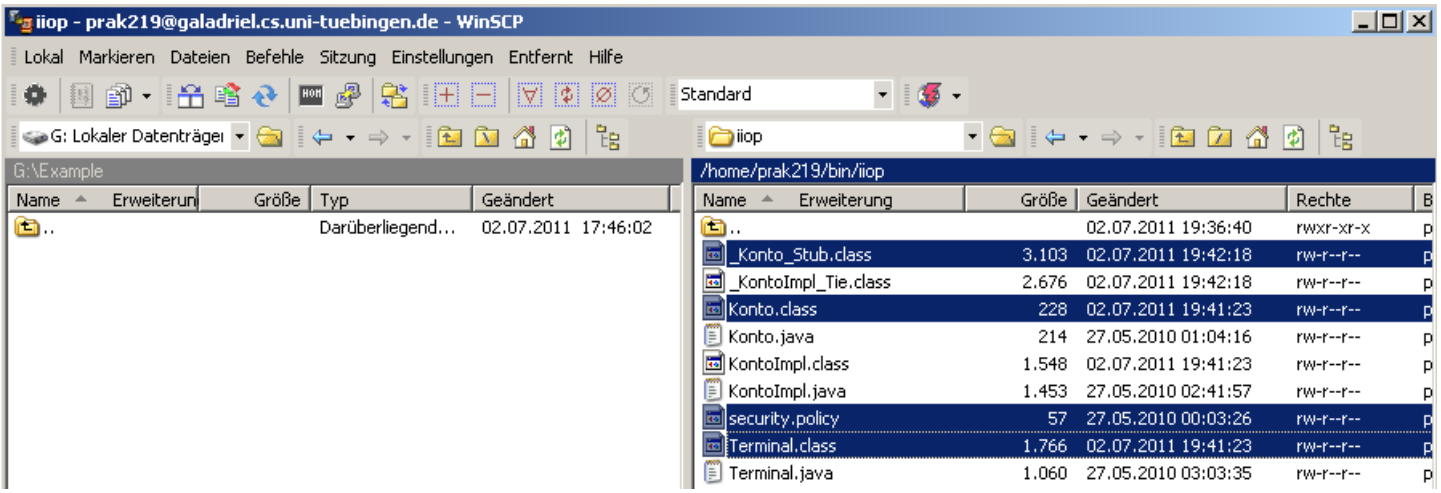
<Enter>, um das Kommando einzugeben.

Damit läuft auch unser Server und wartet auf eine Input Nachricht (über Port 2012).

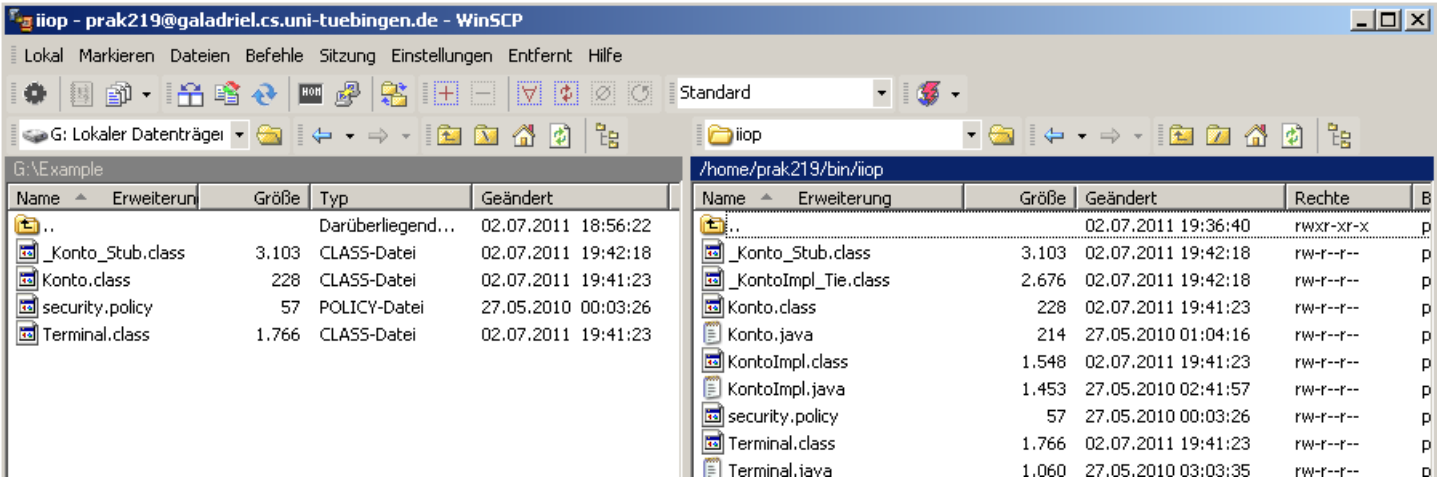
Offnen sie jetzt ein zweites Eingabeaufforderungs-Fenster.

4. Ausführen des Programms

4.1 Download der Client-Klassen



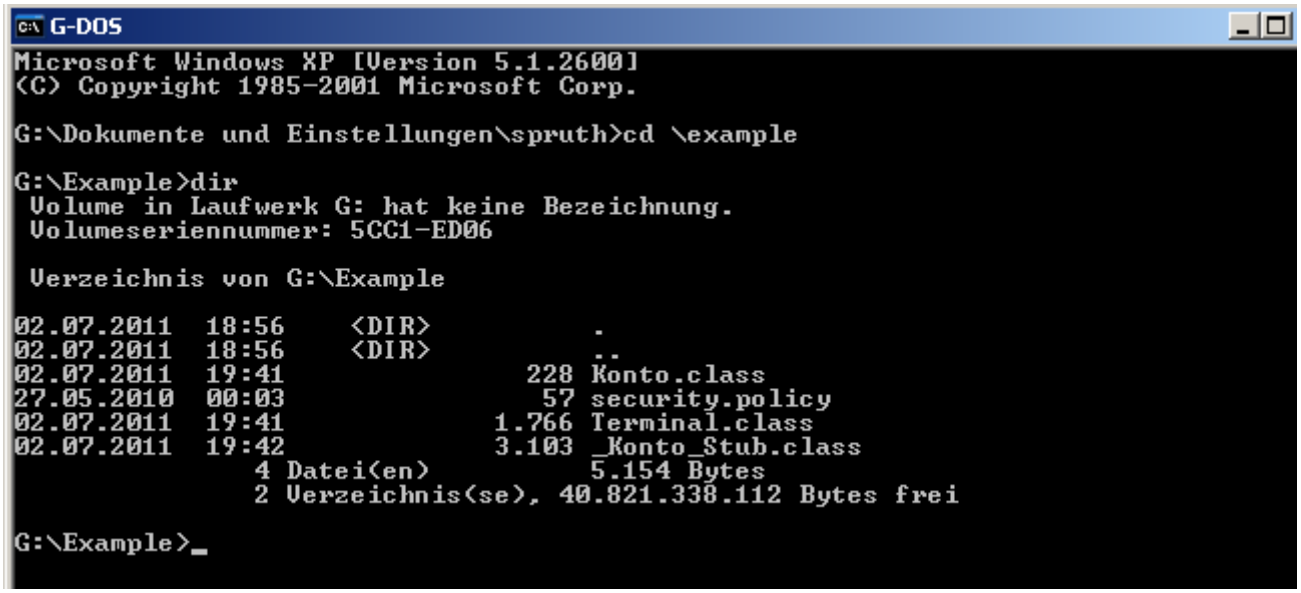
Sie haben auf dem Server Galadriel gleichzeitig auch die vom Klienten benötigten Klassen erzeugt. Kopieren Sie diese `class`-Dateien sowie Ihre `security.policy` auf Ihrem Arbeitsplatzrechner in ein Verzeichnis Ihrer Wahl, in diesem Beispiel „Example“.



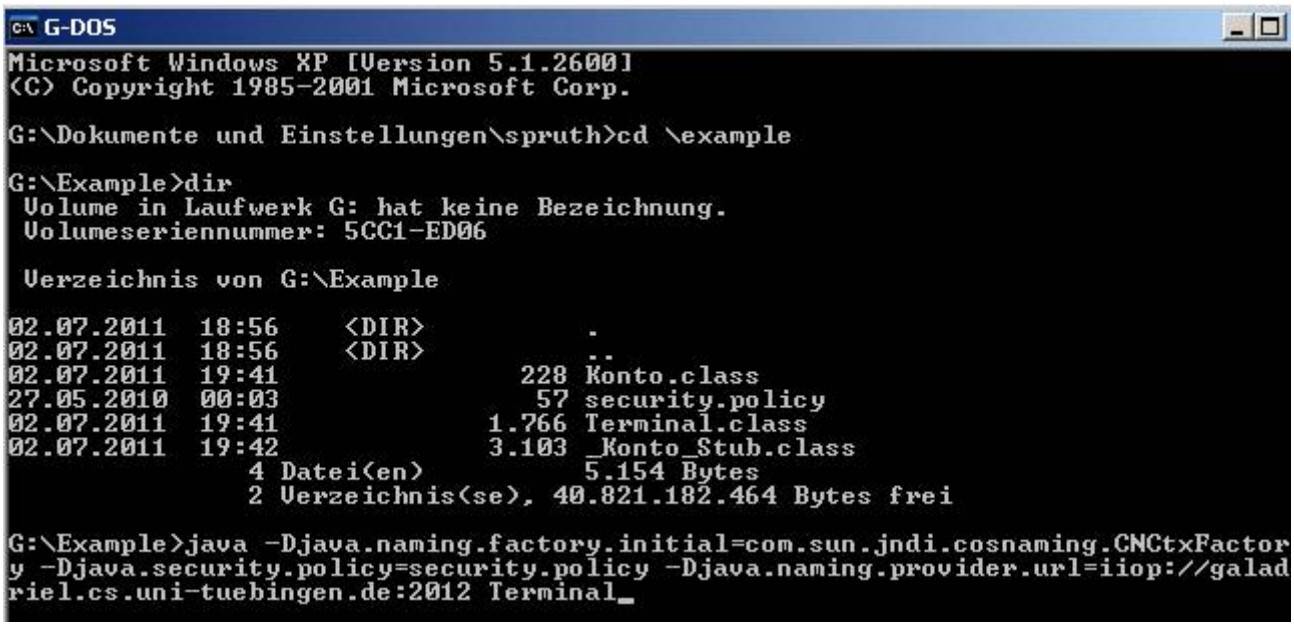
Stellen Sie sicher, dass die `classpath`-Umgebungsvariable Ihres Rechners den Namen dieses Verzeichnisses enthält.

4.2 Aufruf des Servers

Sie haben nun ein ausführbares Java-RMI-Client-Programm auf ihrer Workstation, und ein dazu passendes RMI-Server-Programm auf Galadriel. Sie können nun mit dem Client den Server ansprechen.



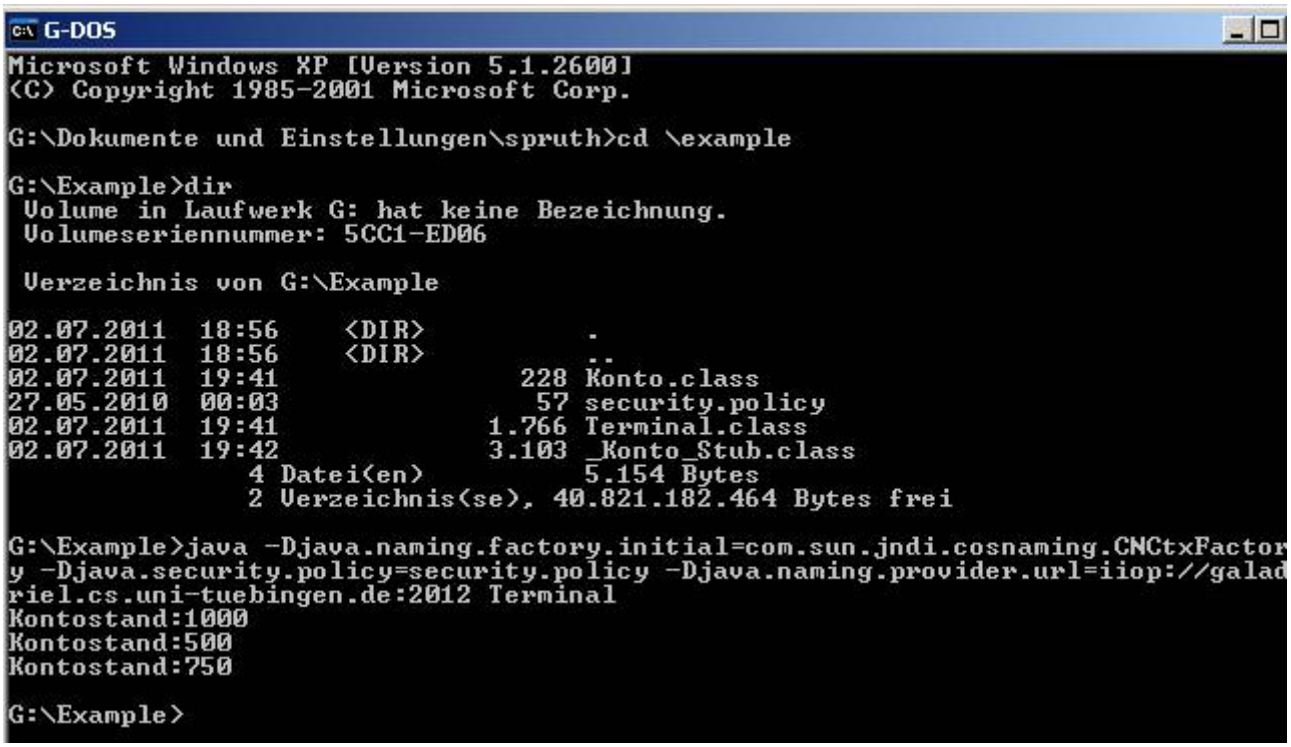
Öffnen Sie auf Ihrem Arbeitsplatz-Rechner ein zweites Eingabeauforderungs-Fenster (zusätzlich zu dem noch immer geöffneten PuTTy-Fenster). Die 4 Dateien befinden sich jetzt auf dem Client in dem Verzeichnis Example.



Mit dem Kommando

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory -
Djava.security.policy=security.policy -
Djava.naming.provider.url=iiop://galadriel.cs.uni- tuebingen.de:2012
Terminal
```

wird der Client gestartet.



Das Ergebnis ist hier zu sehen. Schauen Sie in Ihren Quellcode und verifizieren Sie, dass das Ergebnis Ihren Erwartungen entspricht.

Herzlichen Glückwunsch, you did it! Das Tutorial wurde erfolgreich durchgeführt.

4.3 Herunterfahren des Servers

Wichtig! Zum Abschluss ist jetzt der Server ordnungsgemäß herunterzufahren.

```
prak219@galadriel:~/bin/iiop> java -Djava.naming.factory.initial=com.sun.jndi.co
snaming.CNCtxFactory -Djava.security.policy=security.policy -Djava.naming.provid
er.url=iiop://galadriel.cs.uni-tuebingen.de:2012 KontoImpl
KontoObj bound to registry
KontoServer bereit.
```

Strg+C beendet Ihren Server.

Bitte beenden Sie am Ende wieder Ihren gestarteten tnameserv!
Hierzu ps –aef eingeben.

```
postfix  22496  1386  0 18:21 ?        00:00:00 pickup -l -t fifo -u
prak219  22629     1  0 18:45 ?        00:00:01 tnameserv -ORBInitialPort 2012
root     22804  1298  0 19:52 ?        00:00:00 sshd: prak219 [priv]
prak219  22807 22804  0 19:52 ?        00:00:00 sshd: prak219@pts/1
prak219  22808 22807  0 19:52 pts/1    00:00:00 -bash
prak219  22835 22808  0 19:52 pts/1    00:00:00 ps -aef
prak219@galadriel:~> kill 22629
```

den tnameserv Prozess mit dem kill <pid> Kommando stoppen. Die PID ist hier 22629.

```
prak219@galadriel:~/bin/iiop> exit
logout
```

Das PuTTY-Fenster kann nun geschlossen werden.

Das war es.

Aufgabe: ***Als Abgabe wird eine funktionierende Verison ihrer Anwendung inklusive Quellcode, eine Prozessliste der laufenden Prozesse wenn das Programm arbeitet und eine vollständige Prozessliste, wenn alle Programme beendet sind, sowie ein Screenshot der Ausgabe der Main-Methode der Terminal-Klasse erwartet. Modifizieren Sie den Quellcode, damit die Ausgabe auf dem Bildschirm etwas aussagekräftiger und/oder benutzerfreundlicher aussieht.***

5. Fragen

5.1 Nennen sie mindestens drei Unterschiede zu CORBA

- **RMI ist nur kompatibel zu Java während CORBA ein Standart für die meisten Programmiersprachen ist (solange diese IDL unterstützten).**
- **In CORBA müssen alle im Netzwerk verfügbaren Interfaces über IDL (Interface Definition Language) beschrieben sein. In Java reicht die übliche native Definition von Interfaces aus.**
- **RMI verwendet standardmäßig das JRMP (Java Remote Method Protocol); CORBA verwendet das Internet-Inter-ORB-Protokoll. Dieses ist in den meisten Fällen leistungsfähiger als JRMP.**
- **RMI-über-IIOP ist jedoch ebenfalls möglich (siehe Aufgabe RMI/IIOP).**
- **Mit JRMP können Quellcode und Objekte über das Netz übertragen werden. In CORBA können hingegen nur Datenstrukturen übertragen werden.**

5.2 Nennen sie ein Beispielszenario bei welchem der RMIClassLoader benötigt wird

- **Der RMIClassLoader wird in der Regel benötigt um Klassen aus entfernten Standorten über das Netzwerk zu laden.**
- **Wird beispielsweise eine bestimmte Klasse innerhalb eines Programms benötigt, so kann man diese (manuell) mit Hilfe der Methode *loadClass(URL,String)* in das Programm laden.**

5.3 Welche Vorteile gewinnt man durch die Nutzung von RMI-over-IIOP?

- **RMI-over-IIOP verbindet alle Vorteile von RMI und CORBA.**
- **Es wird das leistungsstärkere IIOP als Übertragunsprotokoll verwendet.**
- **Es können Quellcode und Objekte über das Netz übertragen werden.**
- **Es muss kein IDL verwendet werden um mit Objekten anderer Programmiersprachen zu kommunizieren, da das Interface nativ in Java definiert und mit Hilfe von rmic-idlin IDL überführt werden kann.**

5.4 Wofür steht JNDI? Wozu dient es?

- **Java Naming and Directory Interface.**
- **Namens- und Verzeichnisdienst über das beispielsweise Objekte innerhalb einer CORBA Anwendung gefunden werden können.**
- **In der RMI/IIOP-Aufgabe beispielsweise wird die Klasse KontoImpl mit der Option -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory gestartet. Dabei bedeutet der Wert com.sun.jndi.cosnaming.CNCtxFactory, dass es sich um einen CORBA-Namensdienst handelt.**

6. Anhang

6.1 KontoImpl.java

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

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;

public class KontoImpl extends PortableRemoteObject implements Konto {

int kontostand;

protected KontoImpl() throws RemoteException {
          this.kontostand=1000;
     }

public String getKontostand() throws RemoteException {
     return "Der aktuelle Kontostand ist "+kontostand;
}

public void auszahlung(int betrag) throws RemoteException {
     kontostand=kontostand-betrag;
}

public void einzahlung(int betrag) throws RemoteException {
     kontostand=kontostand+betrag;
}

public static void main (String[] args) throws RemoteException,
NamingException {
     int port = (args.length > 0) ? Integer.parseInt(args[0]) : 2012;


     KontoImpl obj = new KontoImpl();
     String objName = "KontoObj";
     InitialContext INC = new InitialContext();
     if (System.getSecurityManager() == null) {
          System.setSecurityManager (new RMISecurityManager());
     }
     INC.rebind(objName, obj);
     System.out.println ("Konto ready.");
     }

}
```

6.2 Terminal.java

```
import java.net.MalformedURLException;
import java.rmi.*;
import java.rmi.server.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Terminal {

static public void main (String[] args) throws MalformedURLException,
RemoteException, NotBoundException, NamingException {
     String host = (args.length < 1) ? "galadriel.cs.uni-tuebingen.de"
         ://"127.0.0.1" :
     args[0];
     int port = (args.length < 2) ? 2012 : Integer.parseInt(args[1]);
     try {
          if (System.getSecurityManager() == null) {
               System.setSecurityManager (new RMISecurityManager());
          }
          InitialContext INC = new InitialContext();
          Konto obj = (Konto) INC.lookup("iiop://" + host + ":" + port +
              "/" + "KontoObj");
          System.out.println (obj.getKontostand());
          obj.einzahlung(30);
          System.out.println (obj.getKontostand());
          obj.auszahlung(30);
          System.out.println (obj.getKontostand());
          }
          catch(Exception e) {
               System.out.println ("Terminal failed, caught exception " +
                   e.getMessage());
          }
     }
}
```

6.3 Konto.java

```
import java.rmi.*;

public interface Konto extends Remote {
public String getKontostand() throws RemoteException;
public void einzahlung(int betrag) throws RemoteException;
public void auszahlung(int betrag) throws RemoteException;
}
```

6.4 security.policy

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