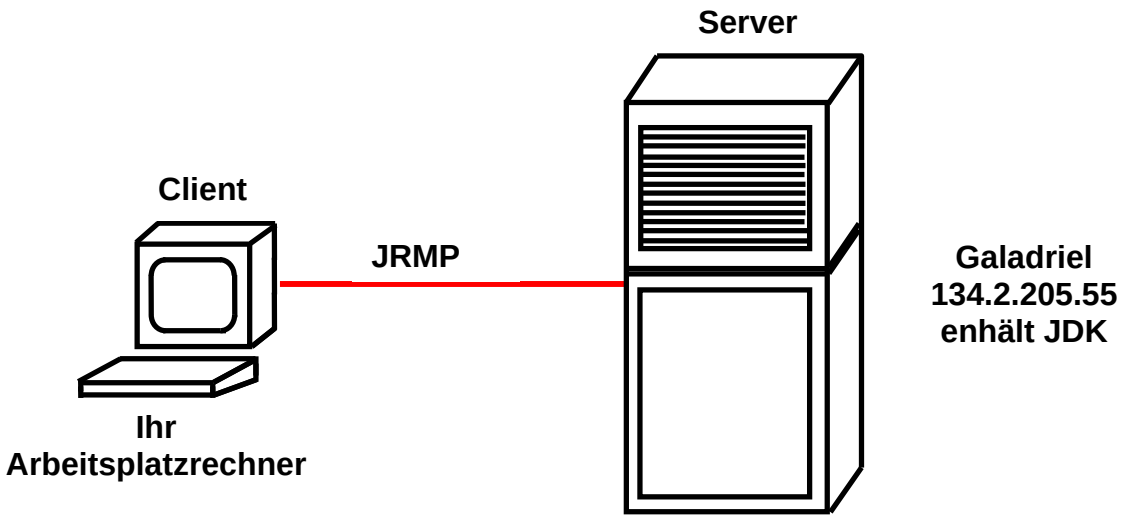
JAVA Remote Method Invocation
JRMP Tutorial

In diesem Tutorial werden Sie ein Java Programm auf einem Klienten (Ihrem PC) installieren und mittels RMI einen Zugriff auf einen Java Server durchführen, der in einer zLinux LPAR auf dem 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.



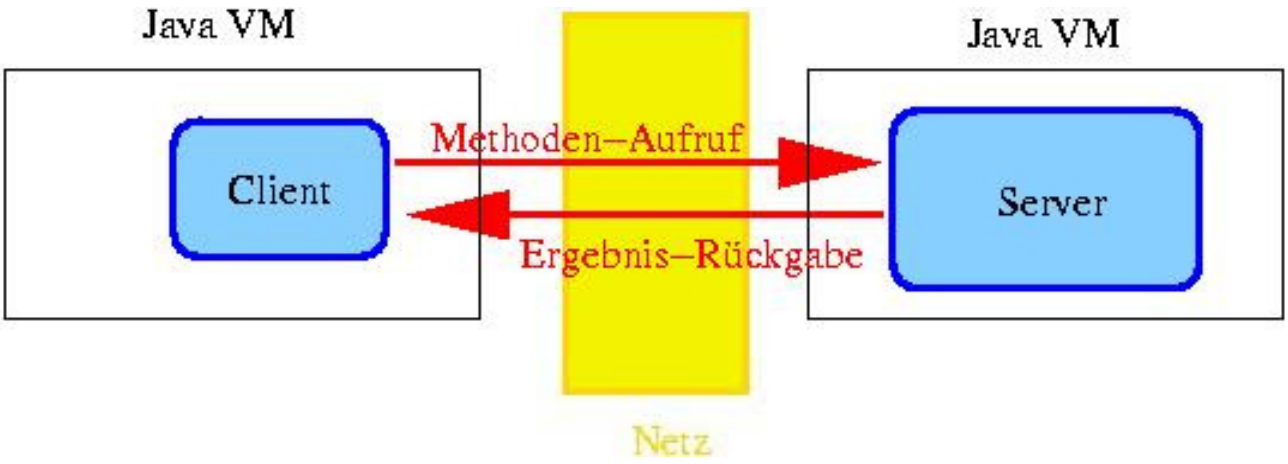
Sowohl auf dem Client als auch auf dem Server ist das Java Development Kit (JDK) installiert. Die Client-Java-Klassen kommunizieren über das Internet mit den Server-Java-Klassen über RMI. Hierfür verwenden sie das Java Remote Message Protokoll (JRMP). JRMP setzt voraus, dass sowohl die Client- als auch die Server-Anwendung in Java programmiert ist.

In einer modernen Implementierung würde man hier einen Web Application Server einsetzen. Die beiden JDKs wären Bestandteil von dessen Servlet-, JSP- und EJB-Infrastruktur. In diesem Tutorial verzichten wir auf diese zusätzliche Komplexität. Wir werden dies in einem späteren Tutorial unter Benutzung des modernen EJB 3.0 Standards nachholen.

Übersicht

1. Das RMI Programmiermodell

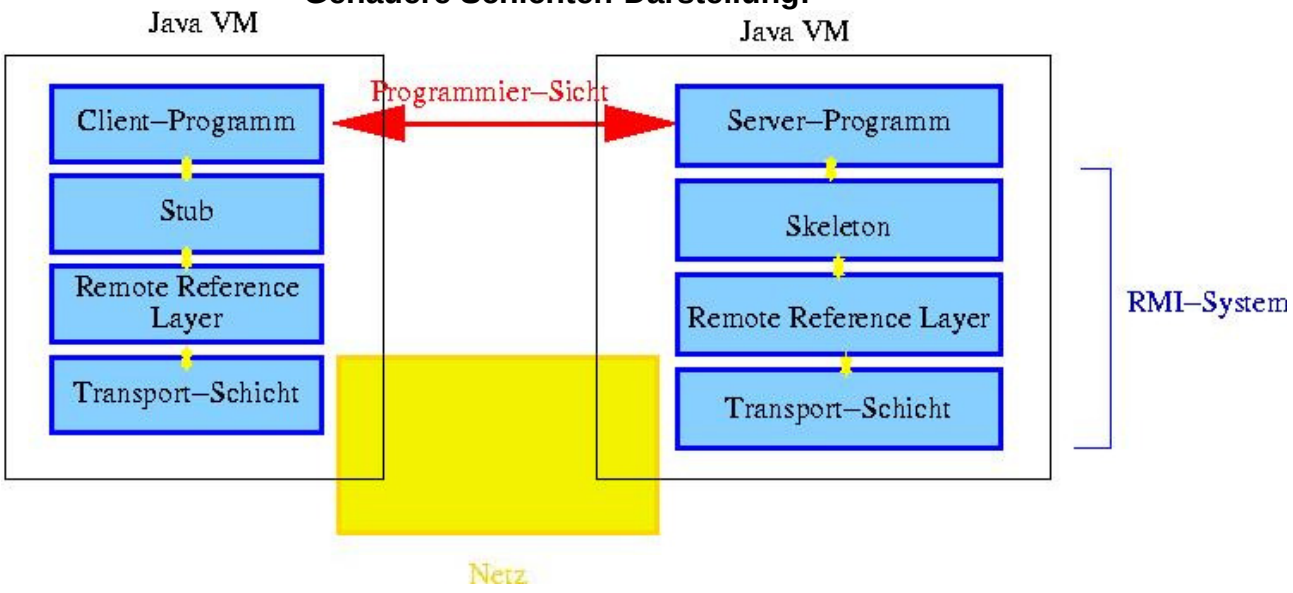
1.1 Java Remote Procedure Call



Normalerweise erfolgen Java-Methodenaufrufe innerhalb einer einzelnen JVM. Java-Objekte in einer JVM können jedoch auch RMI-Methoden von Objekten in einer entfernten JVM aufrufen. Dabei können diese JVMs auf verschiedenen Rechnern im Netz laufen. (Auch unterschiedliche JVMs auf dem gleichen Rechner kommunizieren über RMI.)

Realisierung: Erzeugen eines Stellvertreters (Client-Stub) des entfernten Objekts (Server-Skeleton) in der entsprechenden lokalen JVM. Stub und Skeleton kommunizieren miteinander.

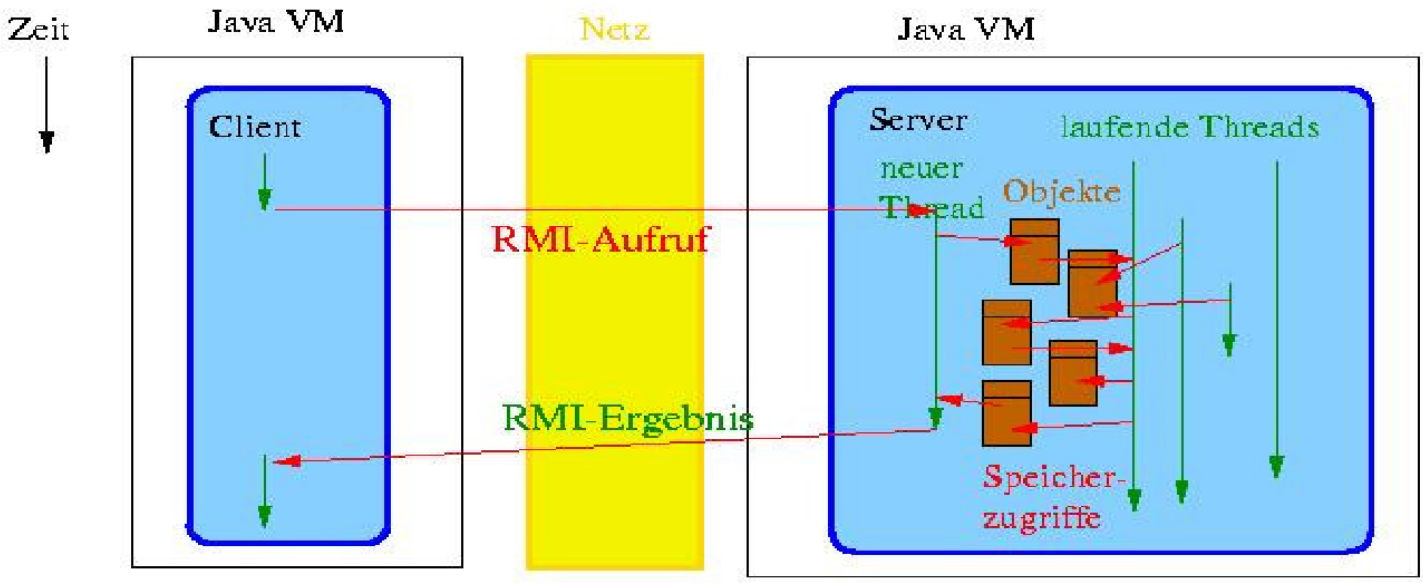
Genauere Schichten-Darstellung:



Der Remote Reference Layer muss entfernte Referenzen (Referenzen auf entfernte Objekte) auf Rechnernamen und Objekte abbilden.

1.2 Threads auf der Server-Seite

Jeder RMI-Aufruf erzeugt auf der Server-Seite (dort, wo die Funktion ausgeführt wird; Skeleton) einen neuen Thread. Das bedeutet, dass dort mehrere solcher Threads gleichzeitig laufen können, zusammen mit evtl. weiteren dauerhaft laufenden Threads des Servers (z.B. Garbage Collector, Compute-Server,...)



Man muss sich also in jedem Fall Gedanken über eine notwendige Synchronisation machen (mit synchronized-Blöcken, ggf. auch mit wait() und notify()).

2. RMI Programmierung

2.1 Übersicht

Auf dem Moodle Server wird im Script z/OS Internet Integration, im Thema 6 „RMI“ unter Teil 4 ein Beispiel für eine einfache RMI Programmierung gezeigt. Allerdings befinden sich hier Client und Server auf dem gleichen Rechner.

Wir werden ein Java-Programm auf einem Klienten (Ihrem PC) installieren und mittels RMI einen Zugriff auf einen Java-Server durchführen, der in einer zLinux LPAR auf unserem z9 Mainframe in Tübingen läuft.

Wir benötigen hierfür 4 Java-Programme, drei davon für den Server

- **Java-Server**
- **Java-Service, auch als Implementation bezeichnet**
- **Java-Interface (des Java Service)**

sowie zusätzlich ein Java-Client.

Wir stellen Ihnen diese Java-Programme vorgefertigt zur Verfügung. Sie dürfen gerne stattdessen eine modifizierte oder andere Version benutzen.

Der Java-Server läuft als Prozess auf einer zLinux LPAR. Normalerweise würden unter einem Java-Server viele Java-Services laufen. In unserem Fall ist dies jedoch nur ein einziger Service. Es ist deshalb in diesem Fall möglich, den Java-Service und den Java-Server zu einem einzigen Java Programm zu kombinieren.

Die drei Java-Dateien, die wir Ihnen zur Verfügung stellen, haben die Namen

Konto.java	**Interface**
KontoImpl.java	**Kombination von Java-Server und Java-Service (Implementation)**
Terminal.java	**Java-Client**

2.1.1 Policy

Das Java-2-Sicherheitsmodell verlangt eine so genannte Policy. Eine Policy besteht aus Regeln, die jeweils Berechtigungen gewähren. Alles, was nicht ausdrücklich in der Policy gestattet wird, ist nicht erlaubt.

Standardmäßig werden Policies in Dateien gespeichert. Die JRE versucht beim Start eines Programms Policy-Dateien auszulesen:

Wir stellen zusätzlich zu den drei Java-Dateien eine einfache Policy mit dem Namen *security.policy* zur Verfügung. Diese Policy ist am einfachsten, aber auch am unsichersten. Der Inhalt der Datei ist:

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

Einen Überblick über mögliche Permissions:
http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html.

Weitere Beispiele für Policy-Files:
http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html#Examples.

Zur Erstellung der Datei java.policy kann auch das GUI-basierte policytool aus dem JDK verwendet werden. Dokumentation:
http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/policytool.html.

2.1.2 Interface java.rmi.Remote

Entfernte Methoden werden durch das Interface java.rmi.Remote definiert: Eine RemoteException kann z.B. auftreten, wenn das entfernte Objekt nicht mehr verfügbar ist oder wenn die Verbindung gestört ist.

2.1.3 Service Implementierung durch java.rmi.server.UnicastRemoteObject

Server-Programme implementieren das Interface Remote und erweitern `java.rmi.server.UnicastRemoteObject`, welches die wichtigsten Methoden für die Verwendung von RMI bereitstellt.

In unserem Beispiel bewirkt der Aufruf

```
public class KontoImpl extends UnicastRemoteObject implements Konto
```

die Implementierung der *Schnittstelle Konto.java*.

2.1.4 rmiregistry

Wie bekommt ein Client Zugriff auf ein Hello-Objekt?

Wir wissen: Stub für Client, Skeleton für Server.

- **Der Server muss den Skeleton unter einem (eindeutigen) Namen bei einer rmiregistry anmelden. Diese rmiregistry muss auf demselben Rechner laufen, auf dem das Remote-Objekt (Skeleton) ausgeführt wird!**
- **Der Client muss**
  1. die URL der rmiregistry kennen, von der er einen Stub des Objekts bekommen kann (z.B. rmi://galadriel.cs.uni-tuebingen.de:2012)
  2. den Namen des Objekts kennen, unter dem der Stub bei der Registry bekannt ist.

2.2 Ihre Aufgabe: Eine verteilte Anwendung mit RMI

2.2.1 Szenario

Eine Bank will ihr System komplett auf Java umstellen. Ein Server soll mindestens ein Konto halten. Von Terminals aus soll Geld eingezahlt, abgehoben und der Kontostand abgefragt werden können.

2.2.2 Code Beispiel

Quellcode Beispiele für drei Dateien

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

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

Server-Seite:

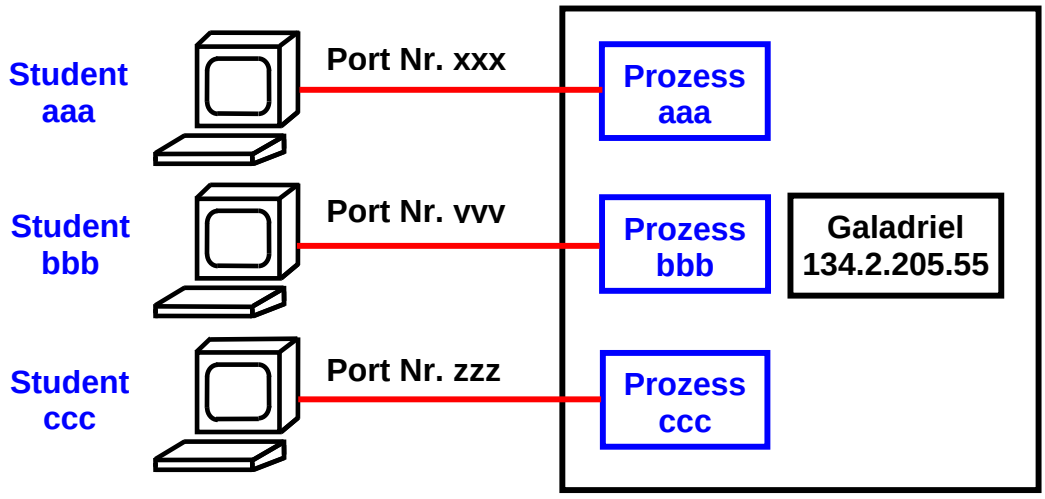
- **`interface Konto extends Remote:`**
  - **beschreibt Methoden zum Einzahlen/Abheben und Abfragen des Kontostandes.**
- **`class KontoImpl implements Konto extends UnicastRemoteObject:`**
  - **Konto zur entfernten Verwendung. KontoImpl hält das aktuelle Guthaben und implementiert die in Konto spezifizierten Methoden.**
  - **Enthält eine main()-Methode, die den RMISecurityManager erzeugt und installiert, falls er nicht schon installiert ist und das entfernte Objekt bei der rmiregistry registiert ist.**

Client-Seite:

- `class Terminal:`
  - holt sich eine entfernte Referenz auf das Konto Objekt.
  - Hält explizit nicht den Kontostand!!! (Das übernimmt die Referenz auf Konto)
  - Hat eine main()-Methode, die eine Reihe von Methodenaufrufen durchführt und eine sinnvolle und lesbare Ausgabe auf die Konsole ausgibt und so die Korrektheit Ihrer Anwendung zeigt.

2.2.3 Port Nummer

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



Eine Anzahl von Studenten wird gleichzeitig dieses Tutorial bearbeiten. Wir benutzen auf Galadriel aber keine Middleware wie z.B. WebSphere. Stattdessen startet jeder Benutzer auf Galadriel seinen eigenen Server-Prozess. Damit braucht jeder Benutzer für seinen Server-Prozess eine getrennte 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 prakxxx 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.

2.2.4 Vorgehen

2.2.4.1 Benötigter Quellcode

Erstellen Sie ein eigenes leeres Verzeichnis auf Ihrem Rechner, z.B. rmi, und erstellen Sie die 3 Java-Dateien:

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

sowie die security.policy-Datei in diesem Verzeichnis.

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.

Die normale Vorgehensweise besteht darin, dass Sie eine passende Entwicklungsumgebung auf Ihrem Arbeitsplatzrechner installieren, dort den Java-Code übersetzen, Skeleton- und Stub-Klassen erzeugen, und diese – soweit sie den Server betreffen – auf diesen kopieren. Da wir nicht mit Servlets und EJBs, sondern mit einfachen Java-Klassen arbeiten, ist ein einfaches Kopieren ausreichend. (Servlets und EJBs würden eine Installation auf einem Web Application Server mittels .jar, .war und/oder .ear Files erfordern.)

Ein einfacher Vorgang? Not really.

2.2.4.3. Exkurs zum Problem der Java Kompatibilität

Java als Programmiersprache entstand in 1995, hat sich aber seitdem nicht richtig stabilisiert. Zahlreiche Probleme bleiben nur unbefriedigend gelöst, besonders im Zusammenhang mit der Isolation von Java Threads und der Zustandsänderung der JVM. Siehe hierzu die Diplomarbeit von Jens Müller, http://www-ti.informatik.uni-tuebingen.de/~spruth/DiplArb/jmueller.pdf

Java ist in mehreren Versionen verfügbar. Version 1.4 erschien 2002, Version 5 erschien 2004, Java Version 6 erschien 2006, Java Version 7 erschien 2011 und Java-Version 8 wird für 2013 erwartet. Statt der Bezeichnung Version 5, 6 oder 7 werden auch die Bezeichnungen 1.5, 1.6 und 1.7 verwendet. Im Gegensatz zu Sprachen wie C++ und Cobol ist die Kompatibilität der einzelnen Versionen untereinander problematisch.

Wegen der Kompabilitäts- und Stabilitätsprobleme führen Firmen wie IBM neuere Versionen nur sehr zögernd ein. Unser Rechner Galadriel verwendet Version 1.4. Wenn Sie heute (2011) eine JDK herunterladen (z.B. von http://www.java.com/de/download/), erhalten Sie Version 1.7.

Wenn Sie auf Ihrem Arbeitsplatzrechner unter Verwendung einer Version 1.7 JDK Klassen für den Server erzeugen, sind diese auf dem Server unter Version 1.4 nicht lauffähig. Die gute Nachricht ist, dass unter Version 1.4 erzeugte Klassen auf einem Rechner mit Version 1.7 ausführbar sind.

Für dieses Problem gibt es zwei Lösungen. Die erste Alternative besteht darin, dass Sie für die Kompilierung auf Ihrem Arbeitsplatzrechner Java Version 1.4 benutzen, und anschließend die Server Klassen nach Galadriel kopieren.

Eine zweite Alternative besteht darin, dass Sie die Kompilierung und Erstellung von Skeleton und Stub Klassen auf Galadriel mit Version 1.4 durchführen, und die für den Klienten erforderlichen Klassen dann von Galadriel auf Ihre Workstation runterladen. Diese Klassen sind dann auf Ihrem Arbeitsplatzrechner auch unter neueren Java Versionen, einschließlich Version 1.7, ausführbar.

Wir verwenden die 2. Alternative. Es steht Ihnen aber frei, mit der hier nicht beschriebenen ersten Alternative zu arbeiten. Falls Sie Schwierigkeiten haben eine passende JDK der Version 1.4 für Ihren Windows Rechner zu finden, können Sie sie unter

http://www.cedix.de/VorlesMirror/Band3/j2sdk-1_4_2_02-windows-i586-p-iftw.exe.

herunterladen. Sie wurde unter Windows XP ausgetestet.

Und wenn Sie dieses Disaster mit den verschiedenen Versionen ärgert, verstehen Sie etwas besser, warum z/OS so großen Wert auf Auf- und Abwärtskompatibilität legt.

2.2.2.4. Stubs und Skeletons mit rmic erstellen.

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

Den rmic-Compiler können Sie mit

```
rmic KontoImpl
```

aufrufen. Als Ergebnis sollten Sie die Dateien

KontoImpl_Skel.class
KontoImpl_Stub.class

erhalten. Die so entstandenen 5 class-Dateien

Konto.class
KontoImpl.class
Terminal.class
KontoImpl_Skel.class
KontoImpl_Stub.class

sind das Ergebnis Ihrer Programmiertätigkeit.

Als Nächstes müssen diese Dateien jetzt installiert, und dann ausgeführt werden.

3. Erstellen der Java Klassen auf dem zLinux Server

3.1 Laden der Klassen auf den zLinux Server

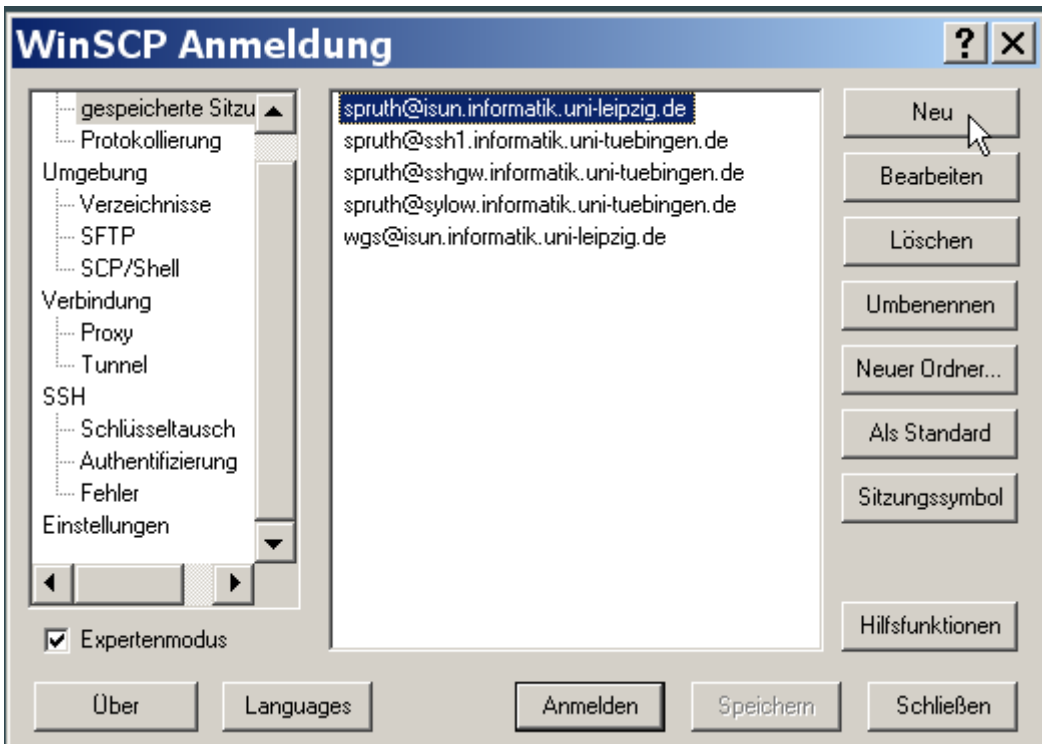
Diese 4 Dateien

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

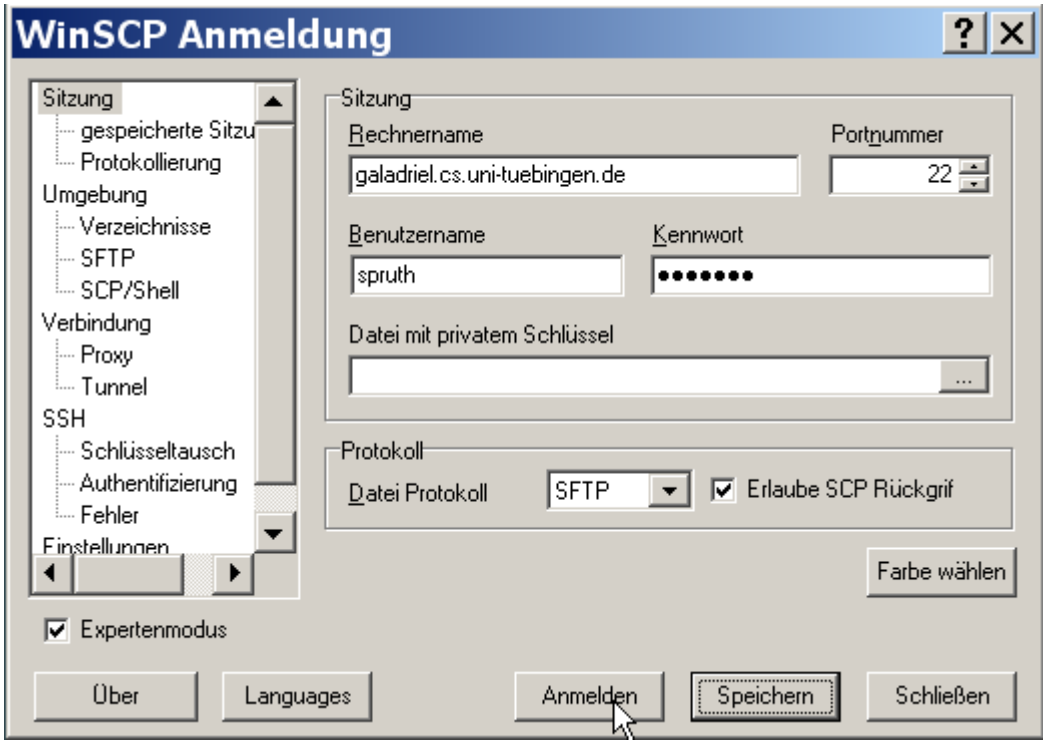
müssen nun auf dem Server galadriel.cs.uni-tuebingen.de (oder 134.2.205.55) geladen werden. Bitte bedenken sie, dass dies eine andere LPAR ist, als die von Ihnen für das letzte Tutorial verwendete LPAR leia.informatik.uni-leipzig.de.

Das Laden geschieht am Einfachsten mit WinSCP (oder FileZilla). WinSCP ist ein Open-Source- SFTP-Client für Windows. Falls noch nicht vorhanden, laden Sie sich WinSCP aus dem Internet herunter, z.B. von http://winscp.net/eng/docs/lang:de, und installieren Sie WinSCP auf ihrem Rechner.

Wir arbeiten 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.



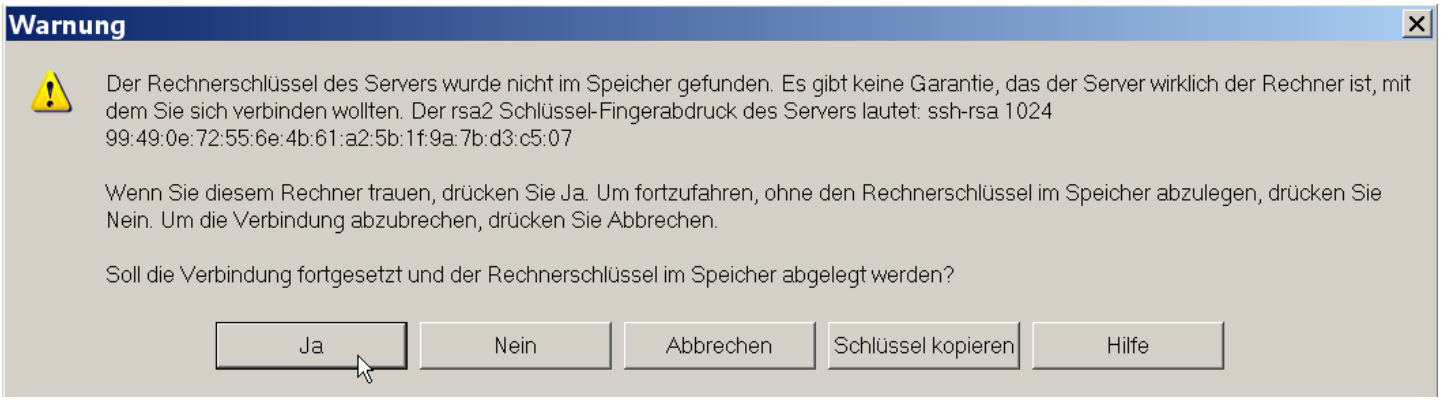
Falls sie WinSCP bereits benutzen erscheinen Ihre bisherigen Verbindungen. Sie müssen für Galadriel eine neue Verbindung erstellen. Klicken sie auf Neu oder New.



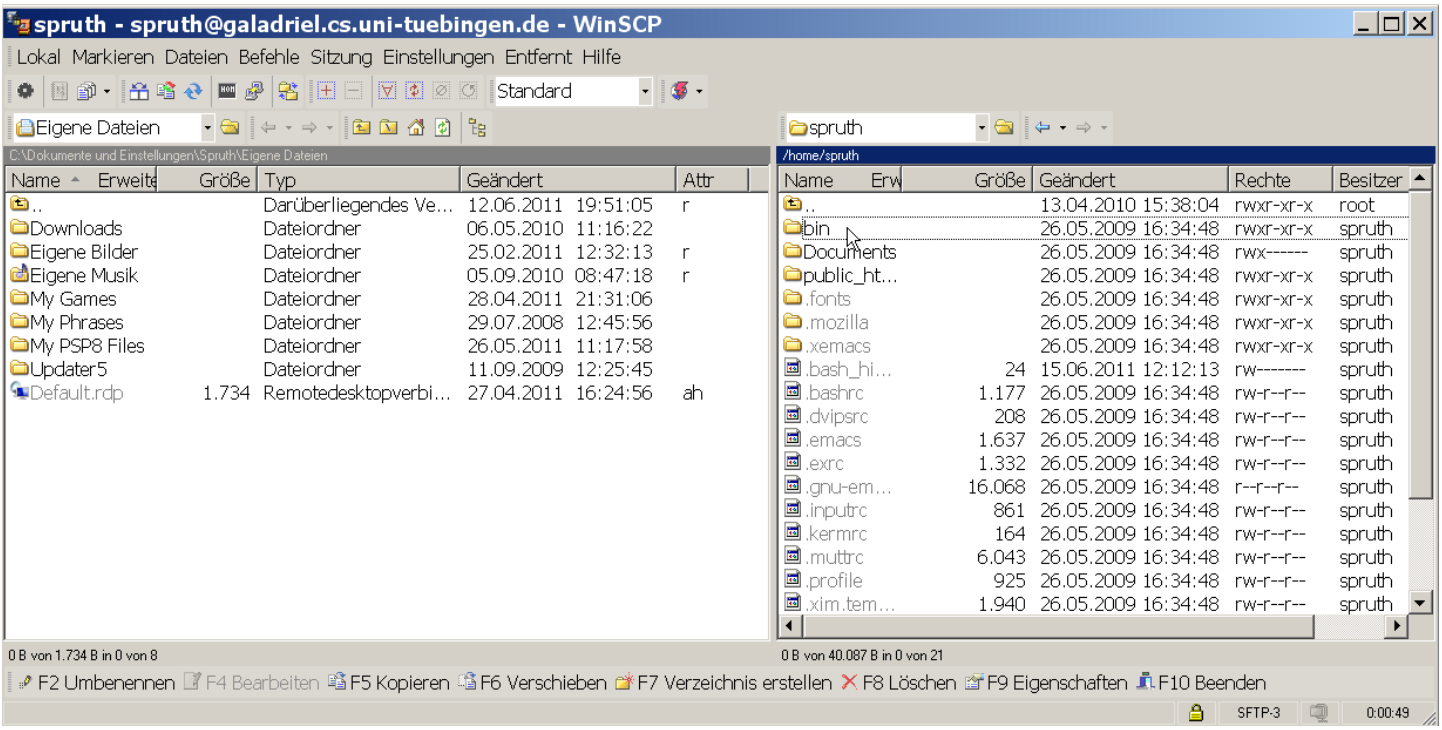
Der Rechnername ist galadriel.cs.uni-tuebingen.de, Port 22.

Wählen Sie SFTP (SSH File Transfer Protocol)

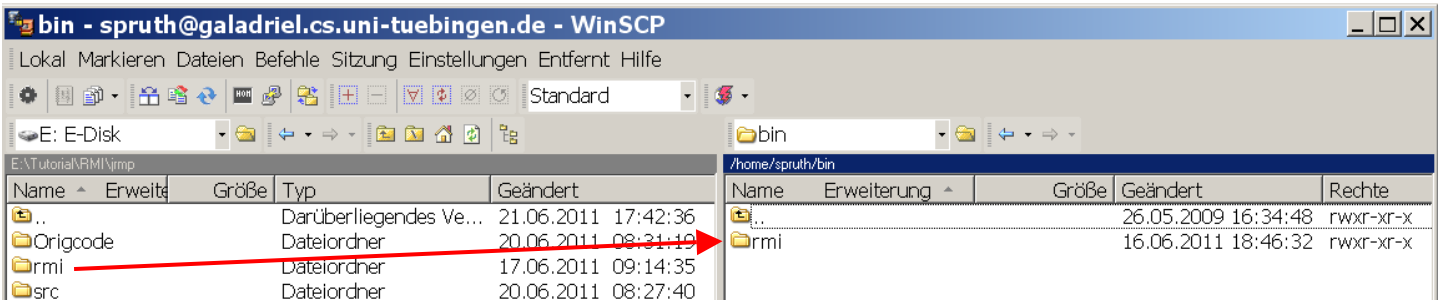
Die in der letzten Übung verwendete Verbindung war zu leia.informatik.uni-leipzig.de, einer z/OS LPAR. Wir haben für Sie eigene User IDs und Passwörter für Galadriel eingerichtet.



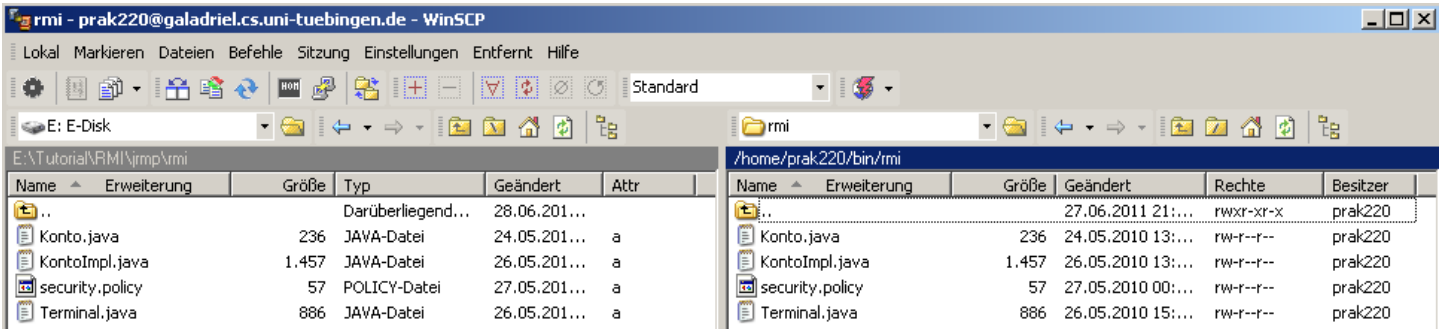
<Enter>



Im rechten Fenster sehen Sie Galadriel, im linken Fenster ihre Workstation. Navigieren Sie im rechten Fenster in das Verzeichnis bin, und im linken Fenster in das Verzeichnis, in dem Sie Ihr Verzeichnis rmi untergebracht haben.



Nun kopieren Sie ganz einfach mit *drag und drop* Ihr Verzeichnis rmi als Unterverzeichnis in das Verzeichnis bin.



Das Ergebnis sieht dann so aus.

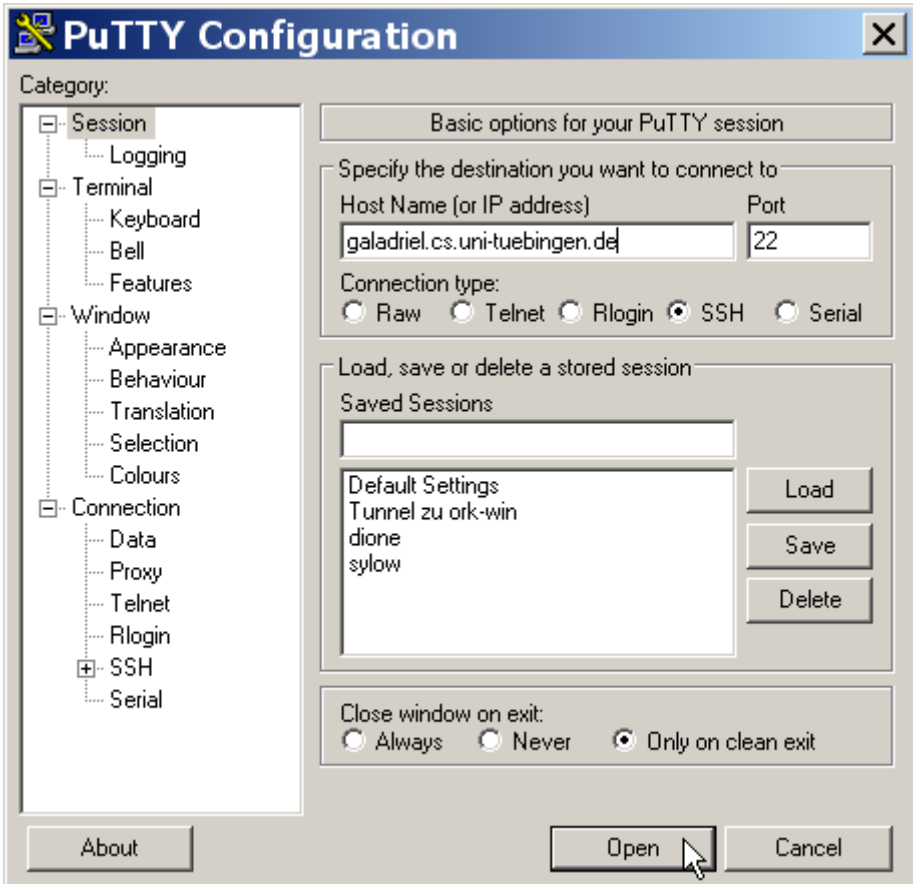
Ok, damit befindet sich der benötigte Quellcode auf Galadriel. Halten Sie das WinSCP Fenster geöffnet.

3.2 Zugriff auf zLinux

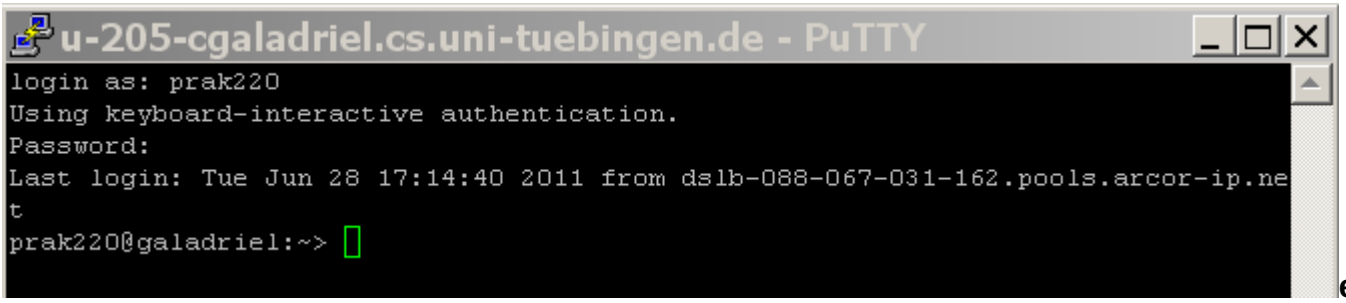
Als nächstes müssen wir den Quellcode kompilieren.

Wir brauchen analog zum 3270-Emulator einen zLinux-Klienten für den Zugriff auf Galadriel. Ähnlich wie beim 3270-Emulator gibt es hier zahlreiche Möglichkeiten. Ein populärer Klient für einen Linux Server ist PuTTY. PuTTY ist ein open source *telnet* und SSH Client für Windows.

Wenn noch nicht vorhanden, downloaden Sie PuTTY von http://www.putty.org/. Rufen Sie PuTTY auf.

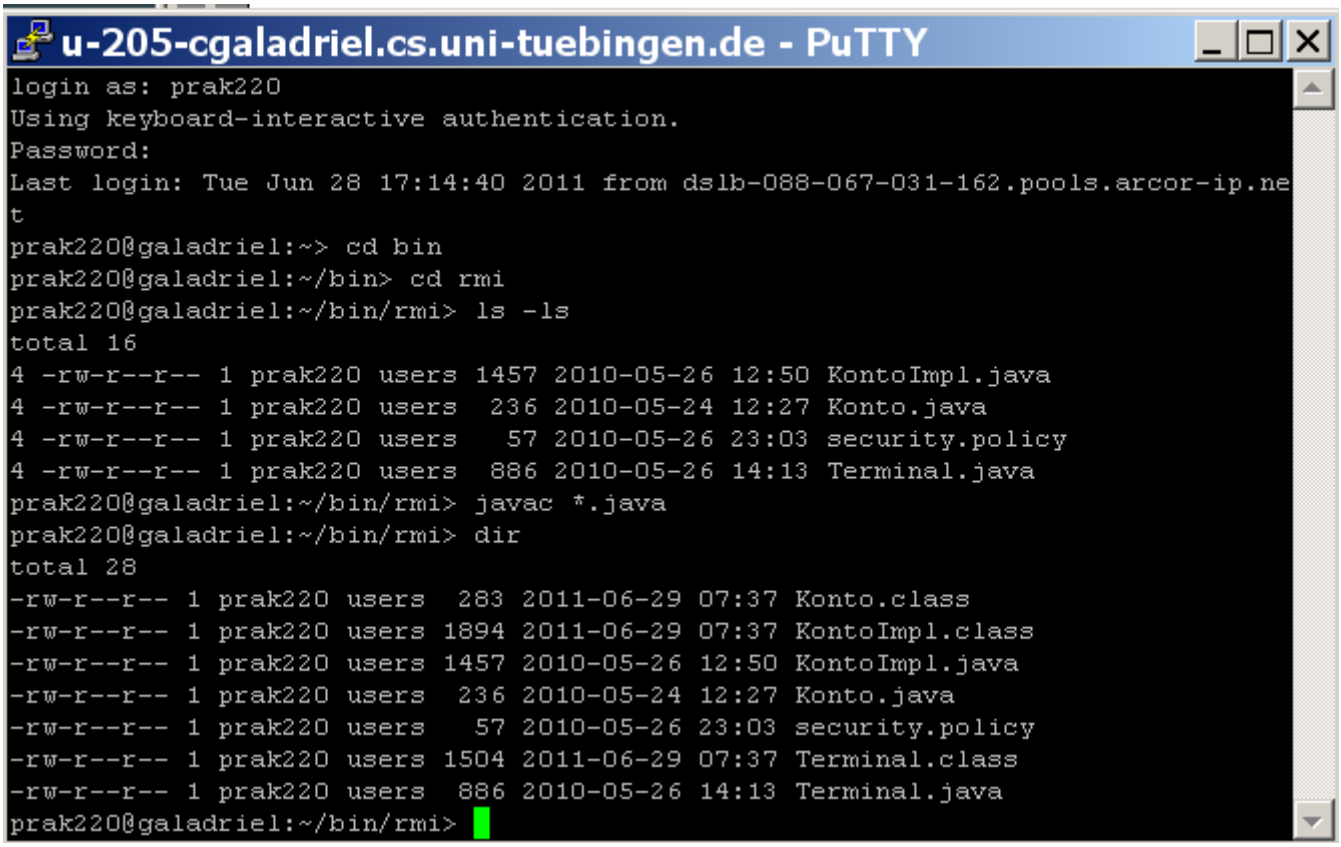


Starten Sie Putty mit der Adresse *galadriel.cs.uni-tuebingen.de, Port 22*.
PuTTY öffnet eine Command-Line-Shell.

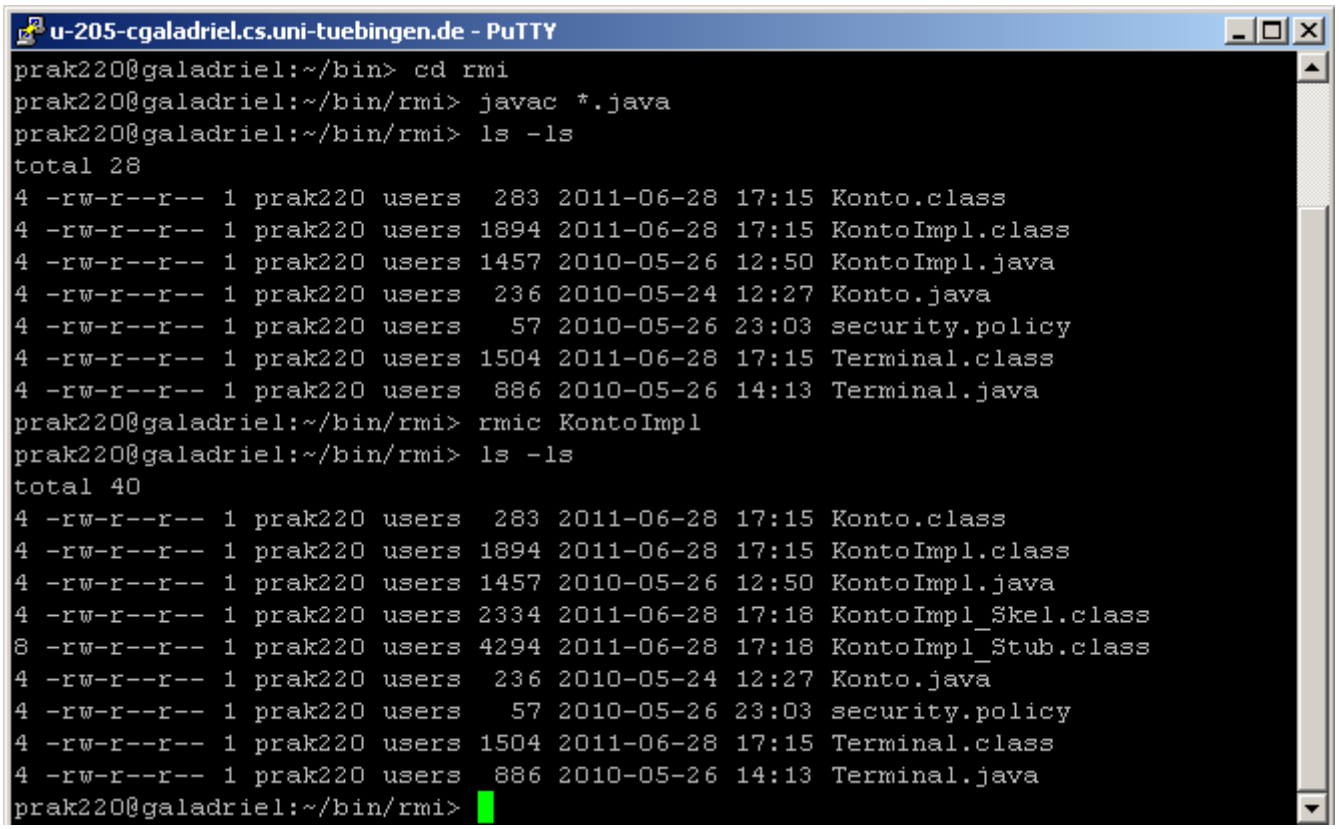


exit

Geben Sie ihre User ID und Ihr Password ein.

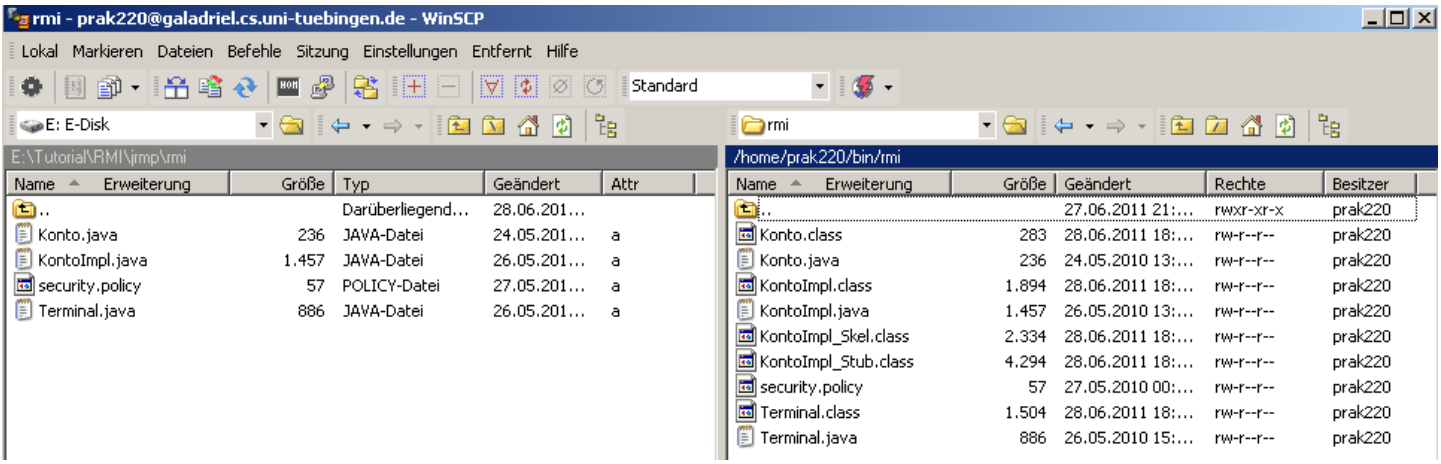


Wechseln sie in das Verzeichnis bin/rmi. Mit javac *.java kompilieren Sie alle java-Dateien.

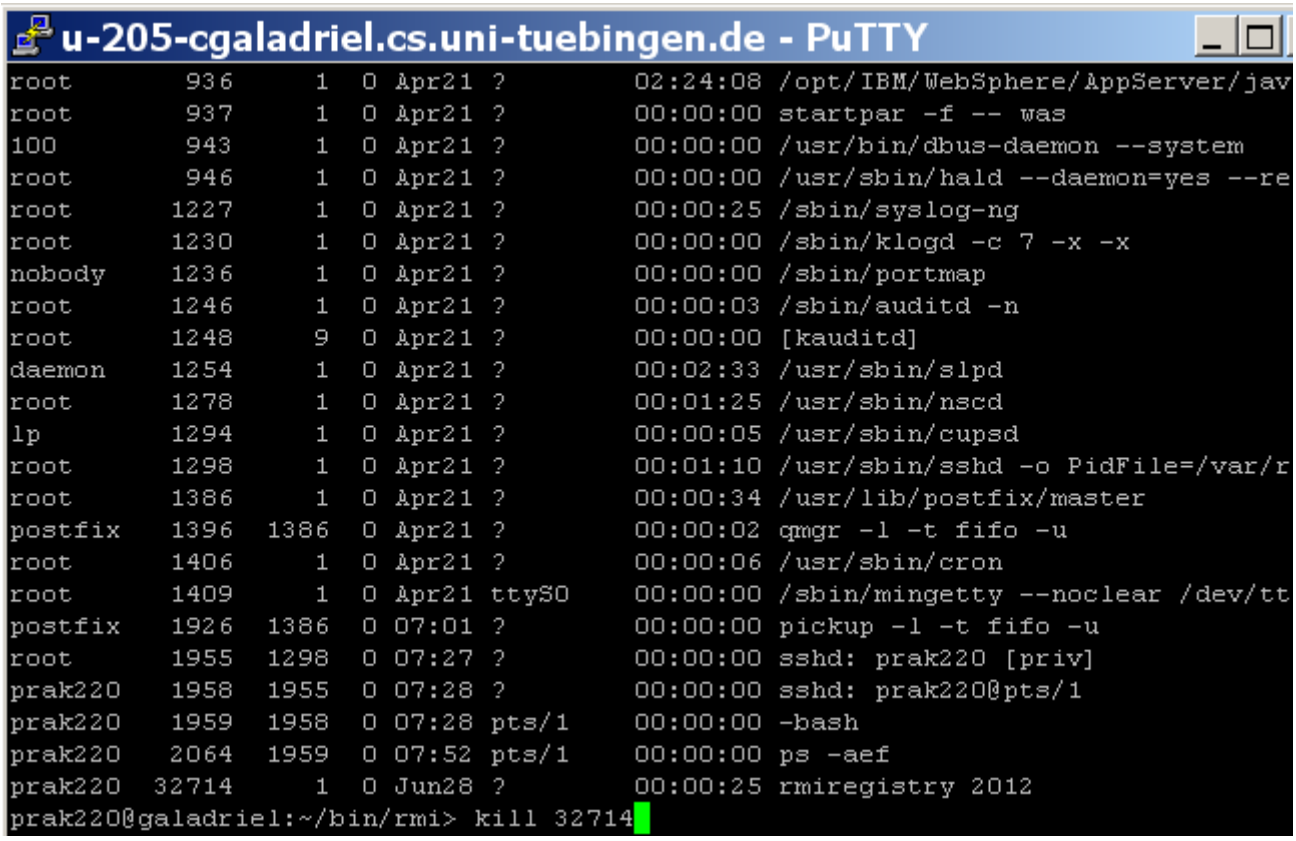


Erstellen sie mit dem Kommando `rmic KontoImpl` Skeleton- und Stub Klassen.

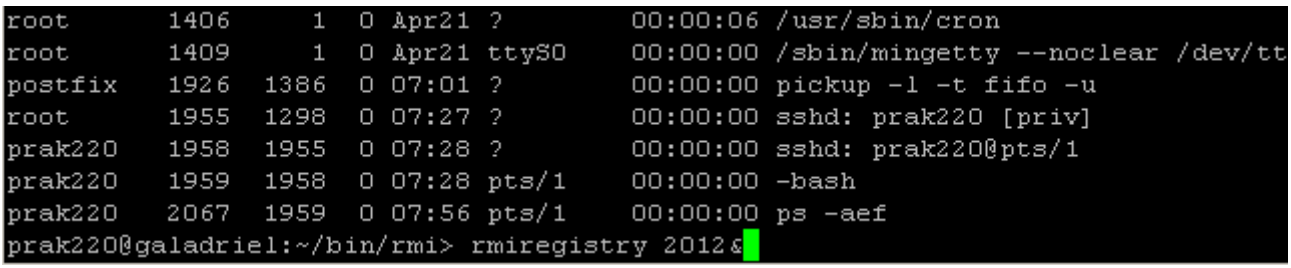
Hinweis: **Skeleton-Klassen werden nicht mehr von neueren RMI-Compiler-Versionen erstellt und benötigt. Sollte also keine KontoImpl_Skel.class nach dem Kompiliervorgang vorhanden sein ist dies kein Fehler.**



Dies ist das Ergebnis auf der Server Seite, was Sie in dem noch geöffneten WinSCP-Fenster sehen können.



Wenn bei der Ausführung des Kommandos eine Fehlermeldung erscheit, kann es sein, dass rmiregistry bereits läuft. Dies kann mit dem Kommando `ps –aef` überprüft werden. Wenn ja, notieren Sie die entsprechende PID und beenden Sie den rmiregistry-Prozess mit dem Command `kill [pid]`; in dem folgenden Beispiel also mit `kill 32714`.



Der Eintrag für den Registry Prozess ist jetzt verschwunden.

3.1.3 RMI Registry

Das“rmiregistry [port]“-Kommando startet eine Remote-Object-Registry auf dem angegebenen Port des verbundenen Servers. Der Aufruf `rmiregistry 2012&` bewirkt, dass unser Serverprogramm über port 2012 erreicht werden kann. Bitte denken sie daran, dass sie hier und bei den folgenden schritten Ihre eigene Port Nr. verwenden.

Das „&“ am Ende des Kommandos bewirkt, dass für die Ausführung ein getrennter Hintergrundprozess eingerichtet wird. Die Ziffer 23556 ist die PID (Prozess-ID) des Hintergrundprozesses.

```
root      1406     1  0 Apr21 ?        00:00:06 /usr/sbin/cron
root      1409     1  0 Apr21 ttyS0    00:00:00 /sbin/mingetty --noclear /dev/tt
postfix   1926  1386  0 07:01 ?        00:00:00 pickup -l -t fifo -u
root      1955  1298  0 07:27 ?        00:00:00 sshd: prak220 [priv]
prak220   1958  1955  0 07:28 ?        00:00:00 sshd: prak220@pts/1
prak220   1959  1958  0 07:28 pts/1    00:00:00 -bash
prak220   2067  1959  0 07:56 pts/1    00:00:00 ps -aef
prak220@galadriel:~/bin/rmi> rmiregistry 2012&
[1] 2068
prak220@galadriel:~/bin/rmi> java -Djava.security.policy=security.policy KontoIm
pl 2012
KontoObj bound to registry, port 2012.
KontoSever ready.
```

Als nächsten Schritt starten wir jetzt auf Galadriel unseren Java-Server.

Hierzu:
`java –Djava.security.policy=security.policy KontoImpl 2012`

Unter RMI läuft eine Client/Server-Verbindung grundsätzlich nur unter einer Sicherheitsvereinbarung, einer Security-Policy. welche sich normalerweise in einer Datei befindet. Mit dem Argument "-*Djava*.security.policy=security.policy" wird spezifiziert, dass die verwendete Security-Policy-Datei den (intellektuell anspruchsvollen) Namen „security.policy“ hat. (Wir hätten die Datei natürlich auch unter einem anderen Namen spezifizieren können.) Die Datei security.policy wird herangezogen, wenn eine Anwendung (hier KontoImpl) über eine Nachricht an Port 2012 aufgerufen wird.

Der Server bestätigt dies mit der Meldung „KontoServer Ready“. Damit ist der Server in der Lage, von einem beliebigen Klienten (hier Ihrer Workstation) über Port 2012 aufgerufen zu werden.

4. Ausführen des Programms

4.1 Erstellen der Client-Umgebung

In unserer Aufgabe wollen wir einen Java-Client mittels RMI mit einem Java-Server verbinden. Hierzu ist es erforderlich dass auf Ihrem Arbeitplatzrechner Java installiert ist. Im einfachsten Fall arbeitet Ihr Client ebenfalls mit einer JDK-Umgebung (an dieser Stelle existieren natürlich viele komfortable Alternativen).

Wir schlagen vor, dass Sie hierfür ein Verzeichnis c:\example einrichten, und in diesem Verzeichnis Ihr Java-Client-Programm ausführen (auch hier können sie nach Ihrer Wahl anders vorgehen). Stellen Sie sicher, dass die *class*- und *classpath*-Variablen richtig gesetzt sind, um eine Java-Programmausführung von diesem Verzeichnis zu ermöglichen. Spezifisch muss die classpath-Variable den Eintrag c:\example aufweisen.

Vielleicht ist es eine gute Idee zu verifizieren, dass Java-Programme korrekt ausgeführt werden.

Hierzu öffnen Sie auf Ihrer Workstation die Eingabeaufforderung. Dieses Fenster existiert zusätzlich zu dem bisher – immer noch offenen – PuTTy-Fenster. Produzieren sie mit einem Editor ein einfaches Test-Programm, z.B.

```
class hallo {
   public static void main(String[] args) {
      System.out.println("Hallo Welt really");
   }
}
```

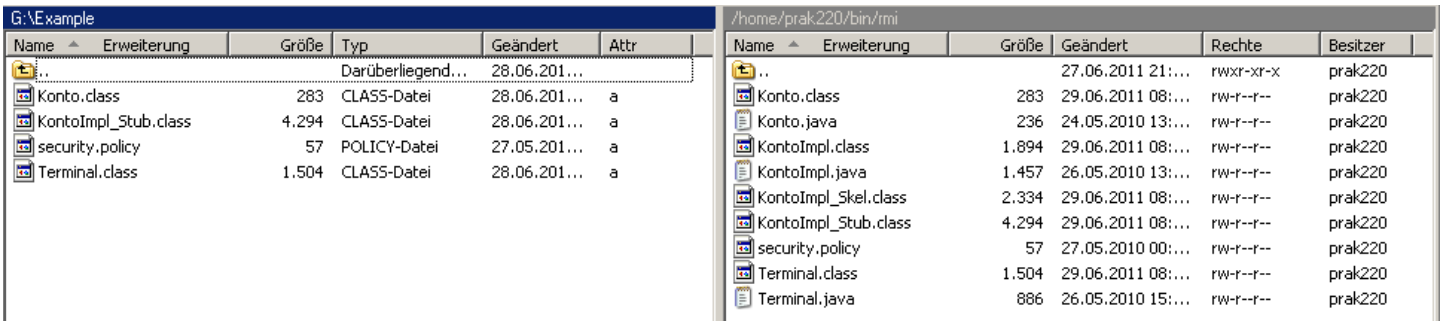
Übersetzen Sie es und führen Sie es aus.

```
C:\>md example

C:\>cd example

C:\example>javac hallo.java

C:\example>dir
 Volume in Laufwerk C: hat keine Bezeichnung.
 Volumeseriennummer: C057-A554

 Verzeichnis von C:\example

18.06.2008  08:14    <DIR>          .
18.06.2008  08:14    <DIR>          ..
18.06.2008  08:14               421 hallo.class
18.06.2008  08:12               117 hallo.java
               2 Datei(en)            538 Bytes
               2 Verzeichnis(se), 16.567.197.696 Bytes frei

C:\example>java hallo
Hallo Welt really

C:\example>_
```

Das Ergebnis sollte so aussehen. Löschen Sie wieder beide Dateien.

4.2 Download der Client Klassen

Beim Erstellen der class-Dateien auf Galadriel mittels *javac* und *rmic* hatten wir die Dateien für den Klienten gleich miterzeugt. Benutzen Sie WinSCP um die Dateien Terminal.class, Konto.class, KontoImpl_stub.class sowie security.policy in das Verzeichnis c:\example zu kopieren.



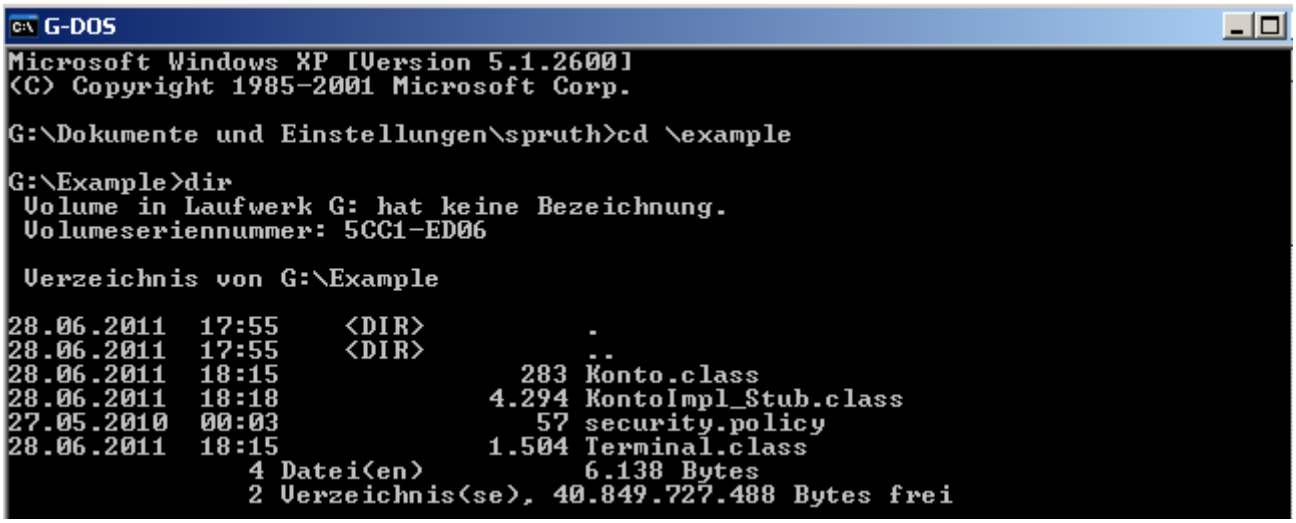
G:\Example

Name	Erweiterung	Größe	Typ	Geändert	Attr
..			Darüberliegend...	28.06.201...	
Konto.class		283	CLASS-Datei	28.06.201...	a
KontoImpl_Stub.class		4.294	CLASS-Datei	28.06.201...	a
security.policy		57	POLICY-Datei	27.05.201...	a
Terminal.class		1.504	CLASS-Datei	28.06.201...	a

/home/prak220/bin/rmi

Name	Erweiterung	Größe	Geändert	Rechte	Besitzer
..			27.06.2011 21:...	rwxr-xr-x	prak220
Konto.class		283	29.06.2011 08:...	rw-r--r--	prak220
Konto.java		236	24.05.2010 13:...	rw-r--r--	prak220
KontoImpl.class		1.894	29.06.2011 08:...	rw-r--r--	prak220
KontoImpl.java		1.457	26.05.2010 13:...	rw-r--r--	prak220
KontoImpl_Skel.class		2.334	29.06.2011 08:...	rw-r--r--	prak220
KontoImpl_Stub.class		4.294	29.06.2011 08:...	rw-r--r--	prak220
security.policy		57	27.05.2010 00:...	rw-r--r--	prak220
Terminal.class		1.504	29.06.2011 08:...	rw-r--r--	prak220
Terminal.java		886	26.05.2010 15:...	rw-r--r--	prak220

Das Ergebnis sieht dann so aus...

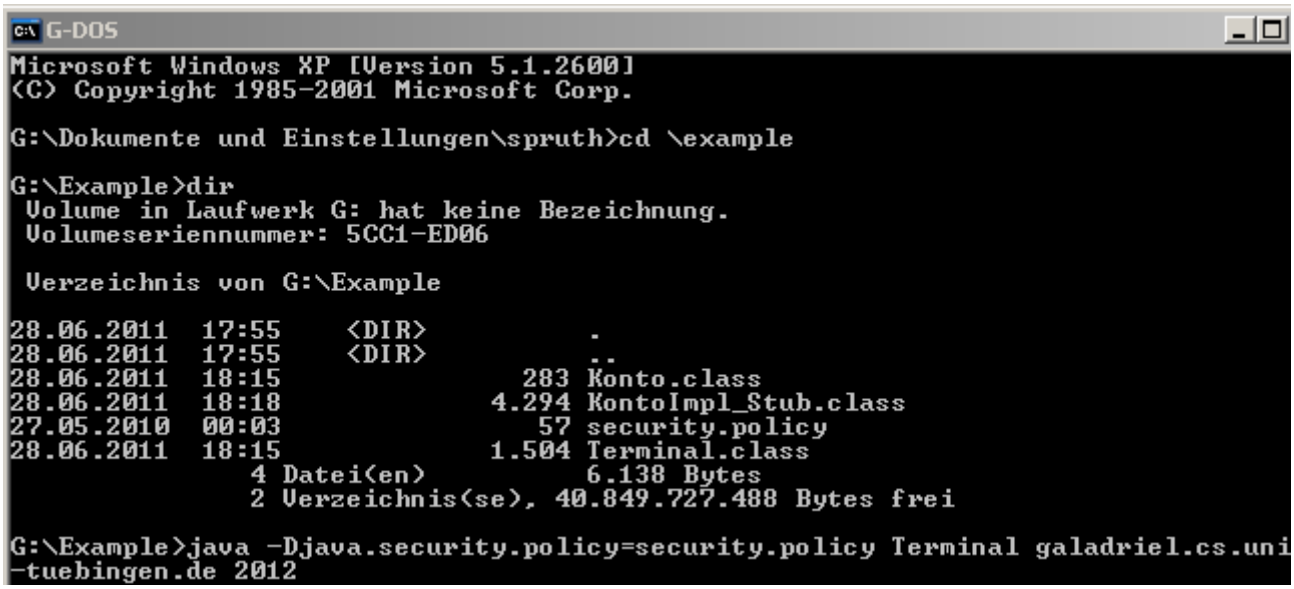


```
G-DOS
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

G:\Dokumente und Einstellungen\spruth>cd \example

G:\Example>dir
 Volume in Laufwerk G: hat keine Bezeichnung.
 Volumeseriennummer: 5CC1-ED06

 Verzeichnis von G:\Example

28.06.2011  17:55    <DIR>          .
28.06.2011  17:55    <DIR>          ..
28.06.2011  18:15               283 Konto.class
28.06.2011  18:18             4.294 KontoImpl_Stub.class
27.05.2010  00:03                57 security.policy
28.06.2011  18:15             1.504 Terminal.class
               4 Datei(en)          6.138 Bytes
               2 Verzeichnis(se), 40.849.727.488 Bytes frei
```

und im DOS-Fenster ebenso.

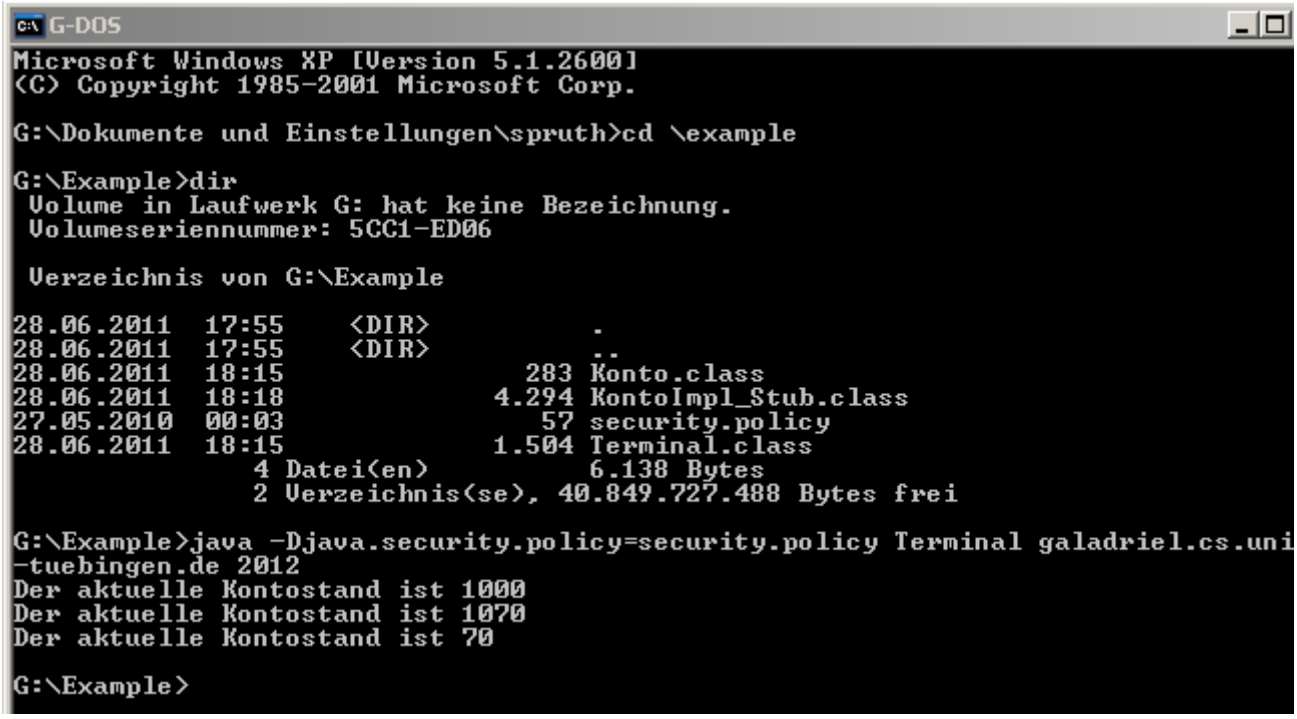
4.3 Aufruf des Servers

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 Klienten den Server aufrufen.



Das geschieht mit dem folgenden Kommando:

java -Djava.security.policy=security.policy Terminal galadriel.cs.uni-tuebingen.de 2012



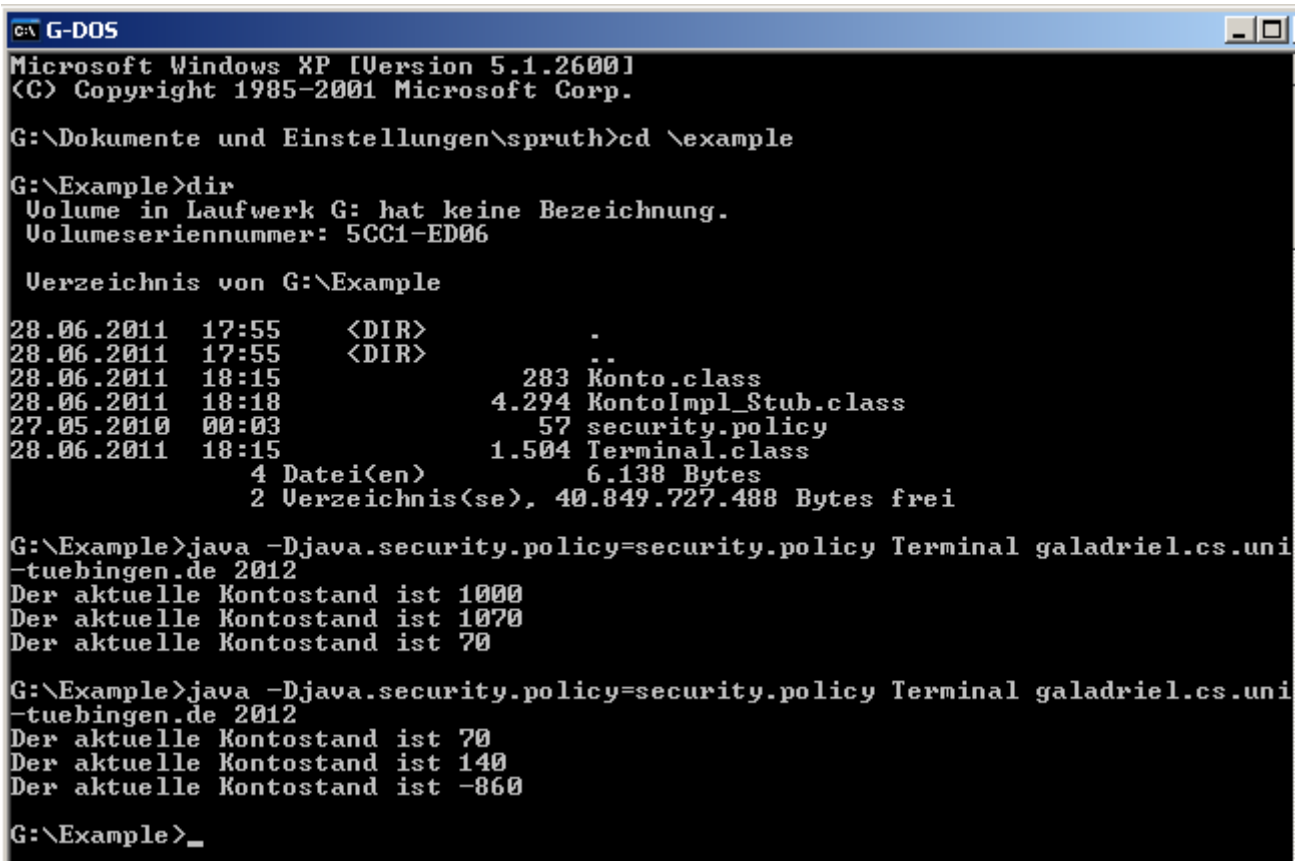
…und sie erhalten diese benutzerfreundliche Antwort.

Am besten schauen sie sich jetzt einmal den Quellcode von

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

an. Verifizieren sie, dass die gezeigte Ausgabe dem Quellcode entspricht.

Aufgabe: ***Erweitern und verbessern Sie die Quellcodes und reichen Sie wie gewohnt einen Nachweisscreenschot ein. Dieser soll ähnlich wie der Screenshot unten die Ausgaben Ihres verbesserten Servers zeigen. Unter diesen Ausgaben muss auch Ihr Vor- und Zuname mit dabei sein als Beleg dafür, dass Sie persönlich die Programmmodifikation vorgenommen haben. Sie könnten z.B. auch noch die Ausgabe des aktuellen Kontostandes durch die Angabe der jeweiligen Währung verbessern, z.B. durch folgende Ausgabe: „Der aktuelle Kontostand von Nils Michaelsen beträgt 1000 EUR“.***

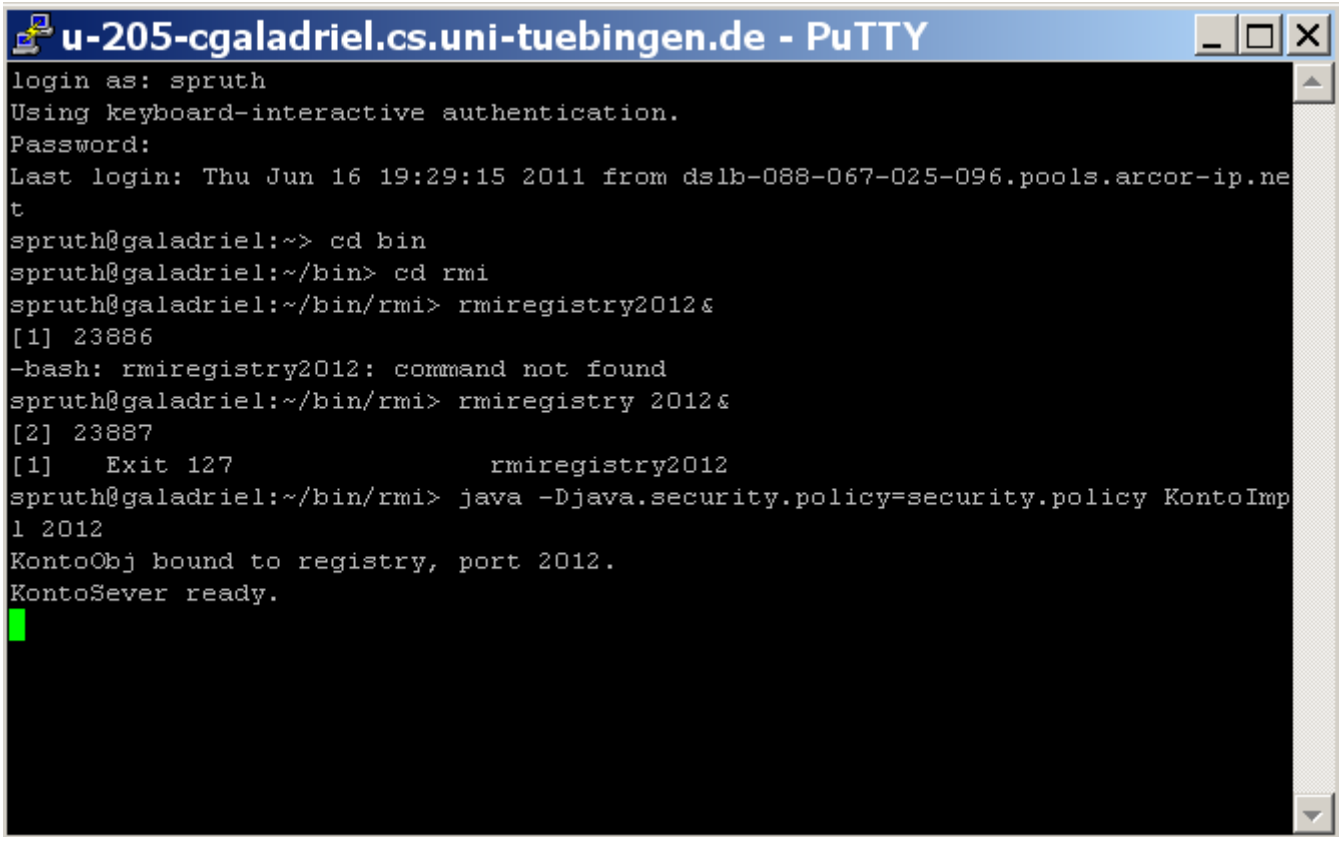


Der hier durchgeführte Methodenaufruf ist zustandslos, ähnlich wie beim Aufruf einer HTTP Seite im WWW. Da Ihr Programm die Variablen des Server-Objektes nicht verändert, erhalten Sie bei einem erneuten Aufruf das gleiche Ergebnis.

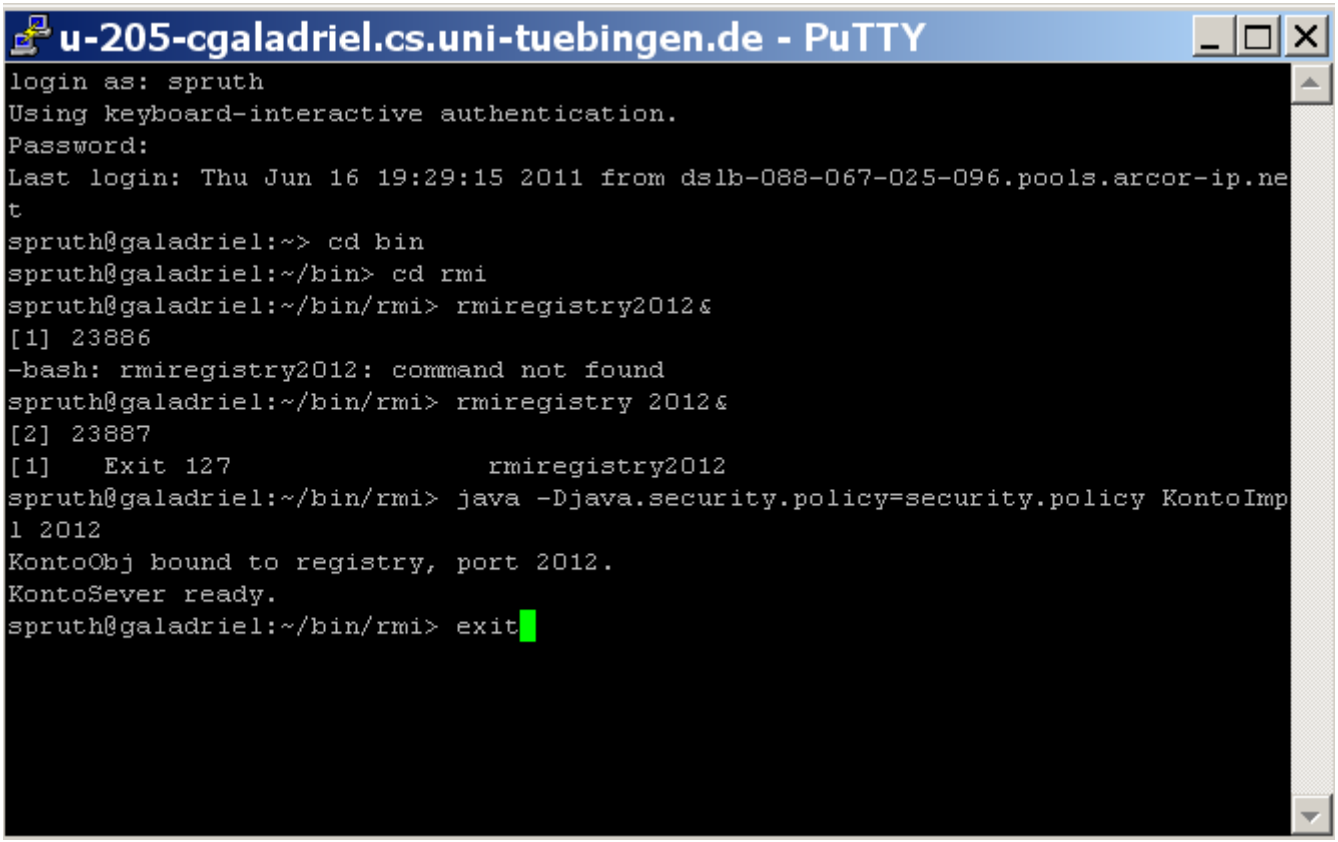
Da die JVM des Java-Servers multithreaded arbeitet, können im Prinzip die Methodenaufrufe mehrerer Teilnehmer von dem von Ihnen erstellten Server gleichzeitig verarbeitet werden. Bedenken Sie aber, dass die anderen Praktikumsteilnehmer alle ihre eigenen Server-Implementierungen mit anderen Ports benutzen.

4.4 Herunterfahren des Servers

Zum Abschluss Ihrer Sitzung sollten sie den auf Galadriel gestarteten Java-Server ordentlich herunterfahren. Gehen Sie wieder in das PuTTY-Fenster.



Auf die primitive Art können sie den Server mit Strg+C terminieren, und...



es ist eine gute Idee, an dieser Stelle den noch laufenden rmiregistry job mit `ps –aef` und `kill [pid]` zu beenden.

Danach Galadriel mit dem exit Kommando verlassen.

Das war es, - herzlichen Glückwunsch zu der erfolgreichen Durchführung des RMI-Tutorials.

Aufgabe: ***Als Abgabe wird eine funktionierende Version 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 Benötigt man mit der J2SE (Java 2 Standard Edition) ab der Version 1.5 ebenfalls den RMI-Compiler rmic? Warum?

- **Nein, der RMI-Compiler wird ab version 1.5 nicht mehr benötigt, da sogenannte dynamische Proxy-Objekte, die zur Laufzeit erstellt werden, die benötigen Informationen bereitstellen. Die Stub- und Skeleton-Klassen werden daher nicht mehr gebraucht.**
- **Werden jedoch ältere Java-Versionen auf Teilen des verteilten Systems verwendet, so muss man dennoch mit Hilfe von rmic die Stub- und Skeleton-Klassen erstellen.**

5.2 Was beinhaltet das Interface java.rmi.Remote? Wieso?

- **Das Interface beinhaltet keine Methoden, da es nur zu Identifikation eines Remote-Objekts dient. Ein Interface, das von java.RMI.Remote erbt, definiert die Remote-verfügbaren Methoden. Jedes Objekt, welches nun dieses Interface Implementiert, stellt daher die implementierten Methoden für Remote-Aufrufe zur Verfügung.**

6. Informationsquellen

Java RMI Home Page:
http://java.sun.com/products/jdk/rmi/index.html

Tutorial:
http://java.sun.com/j2se/1.4.2/docs/guide/rmi/getstart.doc.html

RMI Spezifikation mit ausführlicher Dokumentation:
http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html

RMI Tutorial
http://download.oracle.com/javase/tutorial/rmi/index.html

7. Anhang
Beispielcode

6.1 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;
}
```

6.2 security.policy

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

6.3 KontoImpl.java

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

public class KontoImpl extends UnicastRemoteObject 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 {
     int port = (args.length > 0) ? Integer.parseInt(args[0]) : 2012;

     KontoImpl obj = new KontoImpl();
     String objName = "KontoObj";

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

     Registry registry = LocateRegistry.getRegistry (port);
     boolean bound = false;

     for (int i = 0; ! bound && i < 2; i++) {
          try {
               registry.rebind (objName, obj);
               bound = true;
               System.out.println (objName+" bound to registry, port " +
                    port + ".");
          }
```

```
          catch (RemoteException e) {
               System.out.println ("Rebinding " + objName + " failed, " +
                    "retrying ...");
               registry = LocateRegistry.createRegistry (port);
               System.out.println ("Registry started on port " + port +
                    ".");
          }
     }

     System.out.println ("KontoSever ready.");
     }
}
```

6.4 Terminal.java

```
import java.net.MalformedURLException;
import java.rmi.*;
import java.rmi.server.*;
public class Terminal {

static public void main (String[] args) throws MalformedURLException,
RemoteException, NotBoundException{
     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());
          }
          Konto obj = (Konto) Naming.lookup ("rmi://" + host + ":" + port
               + "/" + "KontoObj");
          System.out.println (obj.getKontostand());
          obj.einzahlung(70);
          System.out.println (obj.getKontostand());
          obj.auszahlung(1000);
          System.out.println (obj.getKontostand());
          }
          catch(Exception e) {
               System.out.println ("Client failed, caught exception " +
                    e.getMessage());
          }
     }
}
```
