In diesen Übungenl:

- Anatomie einer Übung
- Übungsdesignziele
- UML Definition des RMI Beispiel System
- Einfaches Banken System
- RMI Parameter
- RMI Client Callbacks
- Bootstrap Beispiel
- Distributed Garbage Collection
- Serialisierung von Remote Objekten: Server
- Serialisierung von Remote Objekten: Client

3. *RMI - Praxis Übungen*

3.1. Einleitende Bemerkungen

Die Aufgaben liefern Ihnen einen Einblick in die RMI Technologie. Sie zeigen Ihnen, wie Sie Remote Methoden einsetzen, RMI basierte Lösungen entwerfen, implementieren und nutzen können. Die Übungen umfassen auch Clients, damit Sie Ihre Lösungen testen können.

3.1.1. Groblernziele

Nach dem Bearbeiten der Übungen sind Sie in der Lage

- RMI Technologie basierte Server und Clients zu entwickeln
- Sie kennen die grundlegenden Einsatzmöglichkeiten von RMI und
- kennen die grundlegenden Schritte zur Entwicklung einer RMI basierten Lösung

3.1.2. Bemerkungen zu den Übungen

Übungen müssen flexibel sein und unterschiedliche Hilfeebenen anbieten, je nach Voraussetzungen der Studenten.

- **Aufgabenstellung**:
  Einige Studenten werden in der Lage sein, diese Übungen ohne Zusatzinformationen zu lösen. Sie verwenden dazu lediglich die Aufgabenstellung und die Aufgabenliste.
- **Lösungshinweise:**
  Andere werden die Lösungshinweise und die Beschreibung der Lösungsschritte benutzen müssen. Diese folgen jeweils am Ende der Übungseinheit.
- **Lösung:**
  einige möchten sich vielleicht einfach mit dem Stoff vertraut machen und Schntitt für Schritt durch die Lösung geführt werden möchten

Die einzelnen Aufgaben bauen aufeinander auf! Am Schluss haben Sie die Lösung für ein echtes, reales, konkretes Problem.

Für jede Aufgabe stehen Musterlösungen, Lösungshinweise und natürlich die Aufgabenbeschreibung zur Verfügung. Daher könnten Sie auch einzelne Aufgaben überpringen und einfach die Musterlösung der vorherigen Aufgabe als Basis für das weitere Vorgehen verwenden.

Zu jeder Aufgabe werden die Voraussetzungen aufgelistet, so dass Sie sehen können, welche Aufgaben Sie bereits gelöst haben sollten, oder welche Musterlösungen Sie einbinden

müssen. Zudem finden Sie Web-Links auf die API Beschreibungen und jeweils eine kurze Beschreibung der Lernziele dieser Aufgabe.

3.1.3. Übungs Design Ziele

Es gibt drei Arten von Übungen für Sie:

- **"Blank Screen"**
  Sie sehen einen Bildschirm und müssen die Logik dazu entwicklen.
- **Erweiterung**
  Sie müssen ein bereits funktionierendes Beispiel erweitern
- **Korrektur**
  Sie korrigieren fehlerhaftes oder unerwünschtes Verhalten existierender Programme.

Um die Übungen nicht unnötig komplex zu machen, werden Aspekte wie Security in der Regel nicht bearbeitet, ganz im Gegensatz zu realen Programmen.

Daher werden wir auch immer, wenn Applets involviert sind, mit dem Appletviewer arbeiten, da eventuell jemand das Plugin nicht installiert hat.

3.2. RMI Übungen

3.2.1. UML Definition des RMI Beispiel Systems

Diese Übung ist eine Einführung in RMI remote Services mit Hilfe von Java Interfaces.

Lernziele

- Sie kennen die UML Beschreibung der Bankanwendung.
- Sie erstellen eine vollständige Java Spezifikation der System-Interfaces

3.2.2. Einfache Bankanwendung

In dieser Übung starten Sie Ihre erste RMI Anwendung. Sie basiert auf der Bankanwendung, die Sie in der vorherigen Übung zum Laufen gebracht haben.

Lernziele

- Sie lernen die RMI Registry einzusetzen (als separater Prozess)
- starten eines Servers, der remote RMI Objekte unterstützt.
- Implementation eines RMI Client, der remote Services nutzt.

3.2.3. RMI Parameter

Wann immer ein Applikation eine remote Methode aufruft, können Parameter an ein remote Objekt und Rückgabewerte an das aufrufende Objekt übermittelt werden. Diese Übung untersucht, wie RMI einfache Datentypen, normale lokale Objekte und remote Objekte behandelt.

3.2.3.1. Lernziele

- Sie können remote Interfaces definieren und implementieren
- Sie lernen, wie remote Objekte und normale Objekte als Parameter übermittelt werden

3.2.4. RMI Client Callbacks

Bei einigen RMI Systemen macht es Sinn, dass der Server das Client Programm aufruft. Dies kann man mittels Client-seitigen Callbacks.

RMI macht es recht einfach, Client-seitige Callbacks zu implementieren. Man kreiert einfach ein Client-seitiges remote Objekt und übergibt eine Referenz darauf an den Server. Der Server verwendet diese remote Referenz, um die Client-Methode aufzurufen (die Rolle von Client und Server wird umgekehrt).

3.2.4.1. Lernziele

- Sie lernen, wie man Client-seitige remote Objekte kreiert
- Sie erstellen und exportieren remote Objekte, die nicht direkt java.rmi.server.UnicastRemoteObject erweitern

3.2.5. Bootstrap Beispiel

Diese Übung besteht aus drei Komponenten:
1. einem Bootstrap HTTP Server
2. einem Bootstrap RMI Server
3. einem Bootstrap RMI Client

Mit Hilfe dieser Komponenten lernen Sie, wie RMI das dynamische Laden von Klassen und Class Dateien mittels FTP und HTTP Servern unterstützt:
1. als erstes lernen Sie, wie ein einfacher Server Java Class Dateien mit Hilfe des HTTP Protokolls liefert.
2. dann kreieren und starten Sie den Bootstrap RMI Server. Der Server funktioniert analog zu den RMI Servern, die Sie schon gestartet haben. Der Unterschied besteht darin, dass in diesem Beispiel der RMI Client und Server sich im selben Verzeichnis befinden. Der HTTP Server muss die Client Klassen an den Client Rechner liefern.
3. schliesslich kreieren Sie einen einfachen Client. Die einzige Funktion dieses Programms ist, zu starten und dann die eigentliche Client Software aus dem Netzwerk herunter zu laden. Der HTTP Server wird die Java Klassen liefern. Als Beispiel nehmen wir Class Dateien, die Sie aus früheren Beispielen kennen

3.2.5.1. Lernziele

- Sie wissen, wie mit Hilfe von HTTP in RMI Systemen Klassen herunter geladen werden können.
- Sie kennen den groben Aufbau und die Funktionsweise eines HTTP Servers
- Sie schreiben ein RMI Server Programm
- Sie schreiben ein Bootstrap Programm, welches Client Klassen über das Netzwerk lädt
- Sie starten ein Bootstrap Programm und beobachten, wie der HTTP Server anzeigt, welche Klassen angefordert und an den Client geliefert werden.
- Sie lernen die Systemeigenschaft / Property java.rmi.codebase einzusetzen

3.2.6. Distributed Garbage Collection

Der DGC, Distributed Garbage Collector und der lokale Garbage Collector arbeiten zusammen, mit den Ziel den Heapspeicher in jeder Java Virtual Maschine (JVM) der am RMI System teilnehmenden Systeme zu managen.

In dieser Übung habenSie die Möglichkeit mit einem RMI Server und zwei RMI Clients zu experimentieren und zu beobachten, wie der Speicher gemanaged wird.

3.2.6.1. Lernziele

- Sie lernen das Verhalten des Distributed Garbage Collector kennen
- Sie beobachten das Zusammenspiel zwischen lokalem und Distributed Garbage Collectors.
- Sie experimentieren mit unterschiedlichen Java Heap Konfigurationen und beobachten, wie sich diese unterschiedlichen Konfigurationen auf die Funktionsweise des lokalen und Distributed Garbage Collectors auswirkt.

3.2.7. Serialisierung von Remote Objekten: Server

Diese Übung illustriert, wie mit Hilfe des Delegation Patterns ein System kreiert werden kann, welches es eiben Netzwerk-Service gestattet aus einer lokalen JVM in den remote Server zu migrieren. Das Beispiel realisiert eine einfache Delegationsarchitektur. Komplexere Architekturen lassen sich aufbauend auf dieses Beispiel realisieren.

Diese Übung besteht aus zwei Teilübungen:
1. als erstes starten Sie den Server und
2. dann den Client

3.2.7.1. Lernziele

- Sie verstehen, wie RMI die Möglichkeiten remote Objekte zwischen JVMs zu verschieben, einschränkt.
- Sie lernen, wie mit dem Delegate Entwurfsmuster (Design Pattern) diese RMIEinschränkungen umgangen werden können

3.2.8. Serialisierung von Remote Objekten: Client

Das Client Programm benötigt sowohl eine remote Referenz auf einen Service, als auch eine lokale Kopie eines Objekts, welches den Service implementiert.

3.2.8.1. Lernziele

- Sie lernen, wie ein RMI Client aufgebaut sein muss, der entweder eine remote Referenz auf einen Service oder eine lokale Kopie benötigt

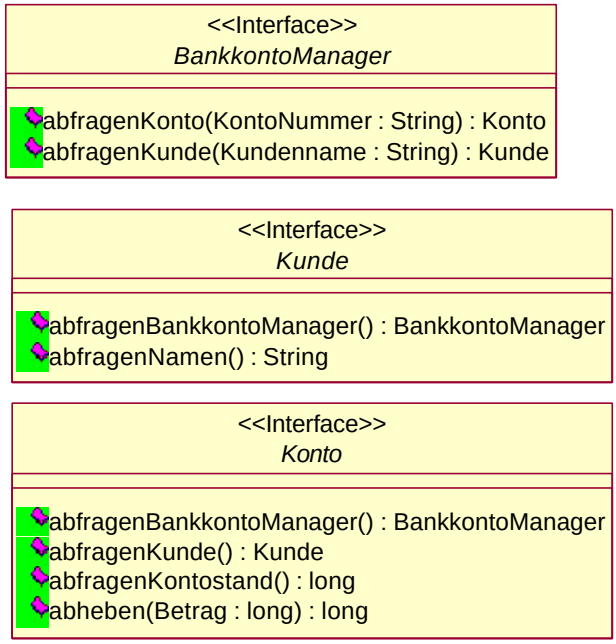
3.3. UML Definition eines RMI Beispiel Systems

Diese Übung zeigt Ihnen, wie das Beispielsystem, eine Bankenanwendung analog zu jener im Skript, aufgebaut ist und zwar als verteilte RMI Applikation mit remote Services, definiert durch Java Interfaces.

Die UML Diagramm sehen Sie nebenstehend.

Die Methoden sind zusammen mit den Signaturen sichtbar.

Die Klassen sind vom Stereotyp her gesehen <<Interface>> und *abstract.*



3.3.1. Programm Skelette

Die folgenden Programm Skelette beschreiben schematisch diese Klassen. Natürlich weiss auch ich, dass Sie mit Rose den Quellcode generieren können! Sie sollten aber zuerst versuchen, die Aufgaben von Hand zu lösen! Die Überprüfung Ihrer Lösung können Sie dann mit Hilfe der generierten Java Quellen durchführen.

BankkontoManager.java

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

public interface BankkontoManager extends Remote
   {
   // definieren Sie eine Methode, die einen Konto remote Service liefert
   // definieren Sie eine Methode, die einen remote Kunden Service liefert
   }
```

Konto.java

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

public interface Konto extends Remote    {

 // definieren Sie eine Methode, die den master Bankmanager liefert
 // definieren Sie eine Methode, die den Client des Bankkontos zurück gibt
 // definieren Sie eine Methode, die denKontostand des Kontos liefert
```

```
 // definieren Sie eine Methode, die einen Betrag vom Konto abhebt

   }
```

Kunde.java

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

public interface Kunde extends Remote {
   // definieren Sie eine Methode, die den master BankManager liefert

   // definieren Sie eine Methode, die den Namen des Kunden liefert
}
```

3.3.2. Aufgaben

In dieser Aufgabe sollten Sie die Notation wiederholen: was ist ein Interface, wie wird ein Interface definiert, wie implementiert?

Zudem sollten Sie

1. die zentralen Klassen der Applikation, die wir als UML Diagramm oben gezeigt haben, kennen lernen
2. die Programmskelette, die Sie oben gesehen haben, vervollständigen

Falls Sie Hilfe benöigen: im folgenden Abschnitt finden Sie einige Hinweise! Nach der Übung haben Sie Java Interface Definitionen für drei remote Services.

3.3.3. Musterlösungen

Musterlösungen finden Sie auf dem Server.

3.3.4. Demonstrationen

In dieser Übung können Sie noch keine lauffähigen Programme erstellen. Die Übung umfasst lediglich das Schreiben der Interfaces.

3.3.5. Hilfe pro Aufgabe

3.3.5.1. Aufgabe 1

Schaien Sie sich das UML Diagramm genau an. Sie finden die UML Diagramme oben in diesem Text oder im Rational Rose Modell.

3.3.5.2. Aufgabe 2

Vervollständigen Sie den Programmcode, mit dem die Interfaces dieses Systems beschrieben werden. Beachten Sie, dass jedes Interface Remote erweitert und dass jede Methode, welche als remote deklariert wird, die Ausnahme RemoteException werfen kann.

3.4. Einfache Bankanwendung

In dieser Übung kreieren Sie Ihre erste selbständige RMI Lösung.

Sie werden die RMI Registry starten, welche die Publikation der remote Dienste verwalten wird, ein Server Programm starten, mit dem der aktuelle remote Service kreiert wird und schliesslich ein Bankkunden Programm kreieren, welches die RMI remote Services nutzt.

3.4.1. Voraussetzungen

Sie kennen die UML Definition des RMI Beispielsystems.

3.4.2. Programmskelette

BankkontoManager.class
BankkontoNichtGedecktException.class
BankKunde.java
Konto.class
KontoImpl.class
KontoImpl_Skel.class
KontoImpl_Stub.class
Kunde.class
KundeImpl.class
KundeImpl_Skel.class
KundeImpl_Stub.class

Diese finden Sie auf dem Server im Übungsverzeichnis.

3.4.3. Aufgaben

Das Vorgehen sollten Sie einhalten und nicht als erstes die Lösungen anschauen!

1. starten Sie das RMI Registry Programm (wir brauchen die Registry später)
2. kopieren Sie alle `.class` Dateien in ein von Ihnen angelegtes Verzeichnis
3. starten Sie den RMI Server, der die Service Objekte beherbergt
4. kreieren und starten Sie ein Programm, welches den exportierten Dienst benutzen wird. Bestimmen Sie den Besitzer des Kontos 4461 und wieviel Geld auf dem Konto liegt.
5. erstellen Sie für diese Aufgabe ein UML Diagramm

Hilfestellungen finden Sie weiter unten. Nach der Übung haben Sie ein laufendes RMI System. Es besteht aus folgenden drei Teilen:

1. der RMI Registry, welche die Referenz auf den remote Service enthält
2. das RMI Server / Host Programm, welches den Service kreiert, diesen bei der Registry anmeldet und auf Client Anfragen wartet.
3. ein Client Programm, welches die Referenz auf das entfernte Objekt bestimmt und dann dessen Services benutzt.

3.4.4. Musterlösungen

Diese befinden Sich im Lösungsverzeichnis. Nochmals: Sie sollten nach Möglichkeit diese Dateien erst nach der Übung konsultieren!

`BankkontoManager.class`

```
BankkontoManager.java
BankkontoManagerImpl.class
BankkontoManagerImpl.java
BankkontoManagerImpl_Skel.class
BankkontoManagerImpl_Stub.class
BankkontoNichtGedecktException.class
BankkontoNichtGedecktException.java
BankKunde.class
BankKunde.java
BankSystemServer.class
BankSystemServer.java
compile_alles.bat
Konto.class
Konto.java
KontoImpl.class
KontoImpl.java
KontoImpl_Skel.class
KontoImpl_Stub.class
Kunde.class
Kunde.java
KundeImpl.class
KundeImpl.java
KundeImpl_Skel.class
KundeImpl_Stub.class
rmic_alles.bat
starten_Client.bat
starten_registry.bat
starten_Server.bat
```

3.4.5. Demo Applikation

Die RMI Registry erzeugt keinerlei Ausgabe, in der `.bat` Datei wird ein echo generiert, dass man das Programm mit CTRL/C abbrechen kann / muss.

Das Server / Host ProgrammServer zeigt an, dass es gestartet wurde und wie es gestoppt werden kann:

```
Server gestartet.
Mit <CR> wird er beendet (System.exit(0)).
```

Die RMI Client-Anwendung BankKunde generiert folgende Ausgabe:

```
Kontostand in USD:
Auf Charlie's Konto befinden sich $600.00
Kontostand in CHF bei einem Wechselkurs von 1.58 :
Auf Charlie's Konto befinden sich SFr. 948.00
Taste drücken, um fortzusetzen . . .
```

Vielleicht gehört das Konto auch Charlie's Tante?
Die gesamte Formatierung der Ausgabe geschieht mit Standardfunktionen aus `java.util`.

3.4.6. Hilfestellungen

1. starten Sie das RMI Registry Programm (wir brauchen die Registry später)

   Das RMI Registry Programm läuft als unabhängiges Programm. Sie könnten die selbe Funktion auch selber oder im Server programmieren. Das ist aber wenig sinnvoll!

   Kreieren Sie ein DOS Fenster und setzen Sie im Fenster den CLASSPATH auf "":
   set CLASSPATH=

Im selben Verzeichnis müssen sich auch die .class Dateien befinden. Später werden wir sehen, wie man mittels Parameter und anderen Methoden .class Dateien aus anderen Verzeichnissen oder Servern nutzbar machen kann.

Das RMI Registry Programm wird, sofern Sie `JAVA_HOME` definiert und das bin Verzeichnis in Ihrem Pfad haben, einfach als

```
rmiregistry
```

gestartet. Die obigen Schritte sind in der Datei `start_registry.bat` zusammengefasst, unter der Voraussetzung, dass das `bin` Verzeichnis im Pfad ist.

2. kopieren Sie alle .class Dateien in das Verzeichnis, aus dem Sie die Registry gestartet haben.

   Sie können auch neue .class Dateien generieren. Dazu stehen Ihnen zwei .bat Dateien zur Verfügung:

   ```
   compile_alles.bat
   rmic_alles.bat
   ```

   Die erste Datei generiert die `.class` Dateien; die zweite die Stubs und Skeletons.

3. starten Sie den RMI Server, der als Host Applikation dient. Diese Applikation wurde bereits für Sie erstellt: `BankSystemServer`. Sie brauchen sie nur noch zu starten. Dafür steht Ihnen eine `.bat` Datei zur Verfügung: `start_server.bat`.

4. nun müssen Sie die Client Applikation entwickeln. Dieses soll die RMI Services, die der Server exportiert hat, benutzen und Details über das Konto 4461 herausfinden: wem gehört das Konto und wieviel Geld befindet sich auf dem Konto?

   Ein Programmskelett für den Client finden Sie im Übungsverzeichnis. Dies sollte Ihnen erlauben, den Client zu vervollständigen und schliesslich auszuführen.

3.5. RMI Parameter

In dieser Übung kreieren Sie einen RMI Service, welcher zweiObjekte an den Client liefert. Sie müssen, auch in der Musterlösung, den Hostnamen `HOST_NAME` anpassen. Der Einfachheit halber habe ich diesen auf `localhost` gesetzt. In der Regel sollte diese Einstellung auch auf Ihrem Notebook funktionieren.

In dieser Übung erweitern Sie das Hallo Interface und die HalloImpl Klasse, um ein Java Objekt (das MessageObjekt) vom Server an den Client zu senden.

Das MessageObjekt wurde so definiert, dass Sie beobachten können, wie eine Klassenvariable auf dem Server hochgezählt wird, während die Variable im übermittelten Objekt konstant bleibt.

Sie finden Hintergrundinformationen zu dieser Übung im Skript im entsprechenden Kapitel.

3.5.1. Voraussetzungen

Sie haben die Übung zum einfachen Bankensystem erfolgreich abgeschlossen.

3.5.2. Programmskelette

```
Hallo.java
HalloImpl.java
MessageObject.java
MessageObjekt.java
RMIClient.java
RMIServer.java
```

3.5.3. Aufgaben

1. ergänzen Sie das Hallo Interface: fügen Sie die Methode `herunterladenMessageObjekt()` hinzu
2. definieren Sie eine Implementation dieser Methode in der Implementation von Hallo: `HalloImpl`.
3. übersetzen Sie alle Programme
4. generieren Sie Stubs und Skeletons für die remote Objekt-Implementation(en)
5. starten Sie die RMI Registry im Verzeichnis, in dem sich die .class Dateien befinden
6. starten Sie den RMI Server in einem DOS Fenster
7. starten Sie den RMI Client in einem DOS Fenster
8. Dokumentieren Sie das Beispiel mit Hilfe von UML

Sollten Sie Hilfe benötigen, dann lesen Sie weiter unten weiter.

3.5.4. Musterlösung

Diese befindet sich im entsprechenden Verzeichnis:

```
compile_alles.bat
Hallo.class
Hallo.java
HalloImpl.class
HalloImpl.java
HalloImpl_Skel.class
```

```
HalloImpl_Stub.class
MessageObjekt.class
MessageObjekt.java
RMIClient.class
RMIClient.java
rmic_alles.bat
RMIServer.class
RMIServer.java
starten_Client.bat
starten_registry.bat
starten_Server.bat
```

3.5.5. Demo Applikation

Die Demo Applikation, die Sie erzeugt haben, liefert je nach dem von Ihnen angegebenen Host eine leicht andere Ausgabe.

Im Folgenden sehen Sie die Ausgabe der Musterlösung. IhreLösung müsste eine ähnliche Ausgabe liefern.

Der Server:

```
CTRL/C beendet den Server
Registry wurde af dem Host localhost an Port 10002 kreiert!
Remote HalloService Implementationsobjekt kreiert
Der Service ist gebunden; er wartet nun auf Kundenanfragen.

MessageObjekt: laufende (Klassen-)Nummer ist  0; Objekt Nummer ist 0
MessageObjekt: laufende (Klassen-)Nummer ist  1; Objekt Nummer ist 1
MessageObjekt: laufende (Klassen-)Nummer ist  2; Objekt Nummer ist 2
MessageObjekt: laufende (Klassen-)Nummer ist  3; Objekt Nummer ist 3
MessageObjekt: laufende (Klassen-)Nummer ist  4; Objekt Nummer ist 4
MessageObjekt: laufende (Klassen-)Nummer ist  5; Objekt Nummer ist 5
MessageObjekt: laufende (Klassen-)Nummer ist  6; Objekt Nummer ist 6
MessageObjekt: laufende (Klassen-)Nummer ist  7; Objekt Nummer ist 7
MessageObjekt: laufende (Klassen-)Nummer ist  8; Objekt Nummer ist 8
MessageObjekt: laufende (Klassen-)Nummer ist  9; Objekt Nummer ist 9
```

Der Client:

```
HalloService Lookup erfolgreich
Antwort des Servers: Hallo, wie geht's!

MessageObjekt: Nummer (Klassenvariable) ist 0; die Objekt Nummer ist 0
MessageObjekt: Nummer (Klassenvariable) ist 0; die Objekt Nummer ist 1
MessageObjekt: Nummer (Klassenvariable) ist 0; die Objekt Nummer ist 2
MessageObjekt: Nummer (Klassenvariable) ist 0; die Objekt Nummer ist 3
MessageObjekt: Nummer (Klassenvariable) ist 0; die Objekt Nummer ist 4
MessageObjekt: Nummer (Klassenvariable) ist 0; die Objekt Nummer ist 5
MessageObjekt: Nummer (Klassenvariable) ist 0; die Objekt Nummer ist 6
MessageObjekt: Nummer (Klassenvariable) ist 0; die Objekt Nummer ist 7
MessageObjekt: Nummer (Klassenvariable) ist 0; die Objekt Nummer ist 8
MessageObjekt: Nummer (Klassenvariable) ist 0; die Objekt Nummer ist 9
Taste drücken, um fortzusetzen . . .
```

Beachten Sie die unterschiedliche Nummerierung! Da eine Kopie des Objekts auf dem Server zum Client gesandt wird, laden beide JVMs ihre eigenen Kopien der Klasse. Die Klassenvariable, als lokale Variable auf dem Server, wird hochgezählt, während diese Variable auf der Kopie auf dem Client konstant bleibt, da jedesmal ein neues Objekt gesandt wird.

3.5.6. Hilfestellungen

1. ergänzen Sie das Hallo Interface: fügen Sie die Methode `herunterladenMessageObjekt()` hinzu
   In der Datei `Hallo.java` müssen Sie die Interface Definition:

   ```
   public MessageObjekt herunterladenMessageObjekt() throws
   RemoteException;
   ```

   hinzufügen!
2. kreieren Sie eine Implementation für die oben definierte Methode in `HalloImpl.java`
   Die Methode liefert eine neue Instanz des MessageObjekts. Der Programmcode könnte etwa folgendermassen aussehen:

   ```
   public MessageObjekt herunterladenMessageObjekt() throws RemoteException
    {
      return new MessageObjekt();
   }
   ```
3. übersetzen Sie alle Dateien für den Server, den Client und speziell die Remote Objekte. Sie finden auf dem Server wieder eine .bat Datei im Verzeichnis der Musterlösung. Diese übersetzt alle .java Dateien.

4. kreieren Sie Stubs und Skeletons:
   auch dazu finden Sie eine rmic Datei, mit deren Hilfe die HelloImpl Stubs und Skeletons generiert werden.
5. starten Sie die RMI Registry, sofern diese noch nicht gestartet wurde.
6. starten Sie den RMI Server in einer DOS Konsole.
   Die .class Dateien müssen sich im Verzeichnis befinden, in dem die Registry gestartet wurde!
7. starten Sie nun im selben Verzeichnis, aber in einem neuen DOS Fenster den Client.

Im Musterlösungsverzeichnis finden Sie .bat Dateen, die Ihnen fast alle Arbeit abnehmen. Sie sollten aber versuchen alles zuerst von Hand zu starten. Vermutlich wird die eine oder andere Fehlermeldung generiert. So lernen Sie viel besser sehr viel über RMI und die Kommunikationsmechanismen als an einem lauffähigen Beispiel!

3.6. RMI Client Callbacks

In dieser Aufgabe bauen wir ein Client Applet, welches auch als Server dient. In der Regel sind Applets Clients. Allerdings kann es in vereinzelten Fällen Sinn machen, dass der Server Callbacks zum Client einsetzt. Dies wollen wir hier realisieren. Der Server wird periodisch das Applet aufrufen und liefert dem Client periodisch die Uhrzeit.

Damit das Applet als RMI Server agieren kann, muss es remote Interfaces exportieren und implementieren. Unser Applet wird das Interface ZeitMonitor implementieren. Das Interface kann von einem ZeitServer genutzt werden.

Der Server kann das Applet nur auf dem Laufenden halten, falls der Server weiss, wo sich das Applet befindet. Daher muss sich das Applet registrieren lassen. Dazu wird die Methode ZeitMonitor des ZeitServers verwendet.

In dieser Übung müssen Sie Interfaces definieren und Server und Applet implementieren.

Sie finden die Theorie dazu im Skript im Kapitel "RMI Client seitige Callbacks".

3.6.1. Programmskelette

```
ZeitMonitor.java
ZeitServer.java
RMIServer.java
RMIApplet.java
```

3.6.2. Aufgaben

1. definieren und übersetzen Sie das ZeitMonitor Interface
2. definieren und übersetzen Sie das ZeitServer Interface
3. vervollständigen Sie die Implementation der Methoden im RMIServer (markierte Stellen: "Ihr Programmcode")
4. definieren Sie die Klasse ZeitTicker im RMIServer
5. übersetzen Sie alle Programme und generieren Sie die Stubs und Skeletons
6. bereiten Sie das Applet vor (stimmt der HTML Code?)
7. starten Sie die RMI Registry, falls diese nicht bereits läuft
8. starten Sie den RMIServer in einem DOS Fenster, im selben Verzeichnis wie die Registry
9. starten Sie das Applet mit dem Appletviewer
10. Dokumentieren Sie diese Beispiel mit Hilfe von UML!

Falls Sie Probleme haben, dann lesen Sie die Lösungshinweise weiter unten.

3.6.3. Musterlösung

```
ZeitMonitor.java
ZeitServer.java
RMIServer.java
RMIApplet.java
```

3.6.4. Demo Applikation

Die Musterlösung im Lösungsverzeichnis wurde mit .bat Dateien versehen, so dass es einfach sein müsste, das Programm / die Programme zum Laufen zu bringen.

Sie können beispielsweise nachdem das Applet gestartet wurde dieses klonen, stoppen und neu starten und erhalten damit genauere Informationen über das Verhalten des Servers. Die folgende Beispielausgabe wird also in Ihrem Beispiel vermutlich nicht genau gleich produziert:

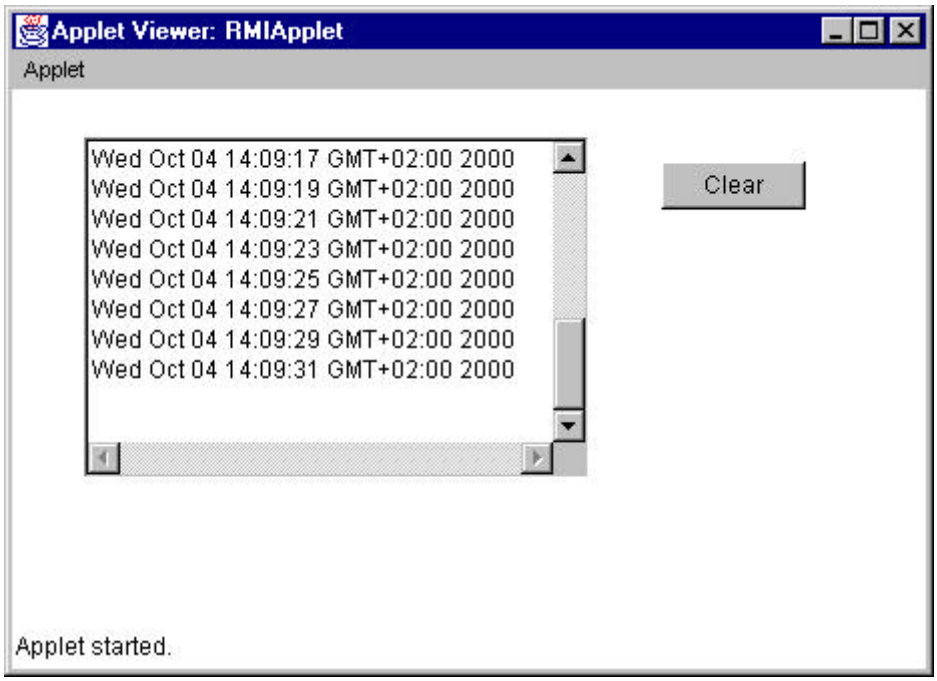
Server:

```
Abbrechen mit CTRL/C
RMIServer: Start
RMIServer: Registry kreiert
RMIServer: Objektbindung abgeschlossen
RMIServer: Warten auf Kundenanfragen
```

Bis hier wurde der Client noch nicht gestartet.
Jetzt starten wir den Client

Client:

```
RMIApplet: init()
RMIApplet: Export des Applet
RMIApplet: DocumentBase
file:/D:/UnterrichtsUnterlagen/ParalleleUndVerteilteSysteme/Kapite
l16 RMI/Übungen/RMICallback/L÷sung/RMIapplet.html
RMIApplet: HostName
RMIApplet: Lookup ZeitService auf dem Server:
rmi://localhost:10005/ZeitServer
RMIApplet: Der Zeit Monitor wurde registriert!
RMIApplet: sagMirDieZeit() Wed Oct 04 14:08:43 GMT+02:00 2000
RMIApplet: sagMirDieZeit() Wed Oct 04 14:08:45 GMT+02:00 2000
RMIApplet: sagMirDieZeit() Wed Oct 04 14:08:47 GMT+02:00 2000
RMIApplet: sagMirDieZeit() Wed Oct 04 14:08:49 GMT+02:00 2000
```



In der selben Zeit werden die Aktivitäten auf dem Server protokolliert:

Server:

... (wie oben)

```
RMIServer: Kundenanfrage liegt vor!
RMIServer: Thread ZeitTicker Konstruktor
RMIServer: Timer gestartet
```

```
RMIServer: Thread ZeitTicker run()
RMIServer: Thread ZeitTicker Zeit:Wed Oct 04 14:08:43 GMT+02:00 2000
RMIServer: Thread ZeitTicker Zeit:Wed Oct 04 14:08:45 GMT+02:00 2000
RMIServer: Thread ZeitTicker Zeit:Wed Oct 04 14:08:47 GMT+02:00 2000
RMIServer: Thread ZeitTicker Zeit:Wed Oct 04 14:08:49 GMT+02:00 2000
RMIServer: Thread ZeitTicker Zeit:Wed Oct 04 14:08:51 GMT+02:00 2000
RMIServer: Thread ZeitTicker stop()
```

Falls mehrere Clients gestartet werden, beispielsweise durch Clonen des Applets:

```
RMIServer: Thread ZeitTicker Zeit:Wed Oct 04 14:10:28 GMT+02:00 2000
RMIServer: Thread ZeitTicker stop()
RMIServer: Kundenanfrage liegt vor!
RMIServer: Thread ZeitTicker Konstruktor
RMIServer: Timer gestartet
RMIServer: Thread ZeitTicker run()
RMIServer: Thread ZeitTicker Zeit:Wed Oct 04 14:12:47 GMT+02:00 2000
RMIServer: Kundenanfrage liegt vor!
RMIServer: Thread ZeitTicker Konstruktor
RMIServer: Timer gestartet
RMIServer: Thread ZeitTicker run()
RMIServer: Thread ZeitTicker Zeit:Wed Oct 04 14:12:49 GMT+02:00 2000
...
RMIServer: Thread ZeitTicker Zeit:Wed Oct 04 14:12:55 GMT+02:00 2000
RMIServer: Thread ZeitTicker Zeit:Wed Oct 04 14:12:55 GMT+02:00 2000
RMIServer: Kundenanfrage liegt vor!
RMIServer: Thread ZeitTicker Konstruktor
RMIServer: Timer gestartet
RMIServer: Thread ZeitTicker run()
RMIServer: Thread ZeitTicker Zeit:Wed Oct 04 14:12:57 GMT+02:00 2000
...
RMIServer: Thread ZeitTicker Zeit:Wed Oct 04 14:13:11 GMT+02:00 2000
RMIServer: Thread ZeitTicker Zeit:Wed Oct 04 14:13:11 GMT+02:00 2000
RMIServer: Thread ZeitTicker stop()
RMIServer: Thread ZeitTicker stop()
RMIServer: Thread ZeitTicker stop()
```

3.6.5. Hilfestellungen

1. definieren und implementieren Sie das ZeitMonitor Interface
   Der ZeitMonitor besitzt eine Methode. Diese Methode wird sagMirDieZeit() genannt. Sie hat einen Parameter, vom Typ Date, und liefert void.
   Vergessen Sie nicht, dass die Methode `java.rmi.Remote` erweitern muss und alle remote Methoden `java.rmi.RemoteException` implementieren müssen.

   Übersetzen Sie die Quelldatei.
   Sie können auch später alle erweiterten und neu erstellten Dateien zusammen übersetzen.

2. definieren und implementieren Sie den ZeitServer
   Dieser Server besitzt eine Methode `registrierenZeitMonitor`. Die Methode verwendet als Parameter den ZeitMonitor und liefert void zurück.
   Vergessen Sie nicht, dass die Methode `java.rmi.Remote` erweitern muss und alle remote Methoden `java.rmi.RemoteException` implementieren müssen.

   Übersetzen Sie die Quelldatei.
   Sie können auch später alle erweiterten und neu erstellten Dateien zusammen übersetzen.

3. vervollständigen Sie die Definition der Methode `registrierenZeitMonitor` im RMIServer.
   Die Klasse RMIServer implementiert das ZeitServer Interface. Sie müssen die Definition der Methode vervollständigen. In der Datei finden Sie Hinweise auf die mögliche Lösung.

4. vervollständigen Sie die Definition des ZeitTickers
   Diese Klasse wird innerhalb des RMIServers definiert. Sie müssen die Klasse vervollständigen und die Definition der `sagMirDieZeit`() Methode und das `ZeitMonitor` Interface definieren bzw. implementieren. Damit wird der Callback zum Applet realisiert!

   Übersetzen Sie die Quelldatei.
   Sie können auch später alle erweiterten und neu erstellten Dateien zusammen übersetzen.

5. generieren Sie Stubs und Skeletons für den RMIServer
   RMIServer hat eine Spezialität: da der RMIServer den ZeitServer erweitert (und diese Remote), kann er das UnicastRemoteObject nicht auch direkt erweitern.

   Das hat zur Folge, dass die Anmeldung, der Export des Objekts explizit selbst programmiert werden muss: UnicastRemoteObject.**exportObject**(((ZeitServer)rmi));
   Normalerweise geschieht dies automatisch, aufgrund der Erweiterung von Remote.

6. Vorbereitung des Applets
   Die HTML Seite für das Applet wurde bereits vorbereitet. Das Applet selbst implementiert den `ZeitMonitor` und muss daher auch manuell exportiert werden.
   Dies geschieht mit der Anweisung: UnicastRemoteObject.**exportObject**(this);

   Übersetzen Sie die Quelldatei.
   Sie können auch später alle erweiterten und neu erstellten Dateien zusammen übersetzen.

   Generieren Sie Stubs und Skeletons für das RMIApplet.
   Sie können auch diesen Schritt am Schluss für den Server und den Client / das Applet gemeinsam ausführen. Dafür steht Ihnen im Lösungsverzeichnis eine .bat Datei zur Verfügung.

7. starten Sie die RMI Registry, falls diese noch nicht läuft
   Im Lösungsverzeichnis steht Ihnen dazu eine .bat Datei zur Verfügung.

8. starten Sie den RMIServer

9. starten Sie das Applet mit dem Appletviewer.
   Auch dieser Schritt wurde im Lösungsverzeichnis vorbereitet.

3.7. Bootstrap Beispiel

In dieser Übung benötigen Sie drei Komponenten:
1. einen HTTP Server (WebServer)
2. den Bootstrap Server
3. den Bootstrap Client

Das Beispiel illustriert, wie Programmcode oder Bytecode dynamisch verschoben werden kann. In unserem Beispiel verwenden wir dazu die Protokolle HTTP und FTP.

Fangen wir mit dem Mini HTTP Server an: dieser implementiert die Grundfunktionalität, die wir für den Dateitransfer, konkret den .class Dateien, benötigen, also dem Transport zwischen RMI Server und RMI Client.

Der Server kann mittels zweier Klassen, die Sun vorbereitet hat, realisiert werden. Sie finden diese auf dem Server:
```
1. ClassFileServer.java
2. ClassServer.java
```

Sie müssen diese übersetzen und in ein separates Verzeichnis transferieren. Die Musterlösung finden Sie im Lösungsverzeichnis und dort im Unterverzeichnis HTTP Server. Sie können die Funktionsweise des Servers testen, indem Sie
1. den Server starten.
   Dazu steht Ihnen eine .bat Datei zur Verfügung (`start_http.bat`)
2. mit Telnet auf diesen Port eine Verbindung aufbauen
   ```
   C:\>telnet localhost 2002
   ```
   Der Server muss sich mit einer Fehlermeldung bemerkbar machen, sofern Sie einfach die Leertaste drücken. Das Server Protokoll sieht beispielsweise folgendermassen aus:
   ```
   ClassFileServer: Start
   ClassServer: Konstruktor
   ClassServer: neuer Listener Thread
   ClassFileServer: gestartet
   ClassServer: Listen Thread Start
   ClassServer: neuer Listener Thread
   ClassServer: Listen Thread Start
   ClassServer: HTTP Fehlermeldung
   ClassServer: finally
   ClassServer: neuer Listener Thread
   ClassServer: Listen Thread Start
   ClassServer: HTTP Fehlermeldung
   ClassServer: finally
   ```

Im Rahmen von RMI werden wir als "Verzeichnis" (zweiter Parameter des HTTP Servers) jenes angeben, in dem sich der RMI Server befindet.

Als nächstes kreieren wir den RMIServer. Dazu verwenden wir einige der Dateien aus dem obigen Beispiel "RMI Parameter":
```
Hallo.java
HalloImpl.java
MessageObjekt.java
```

Übersetzen Sie nun diese Dateien. Die Musterlösung enthält eine .bat Datei, die diesen einfachen Schritt für Sie erledigt. Die .class Dateien lassen wir im speziellen Server Verzeichnis (RMIServer).

Als nächstes kreieren Sie den BootStrap Server.
Der Server ist ähnlich zum RMI Server, den Sie schon zum Laufen gebracht haben. Ein Unterschied besteht darin, dass dieser Client Code verwendet, der im Verzeichnis des Servers steht, also unter Umständen auf einem entfernten Host. Die Dateien sind bereits für Sie generiert worden. Sie müssen aber die Batch Dateien Ihren Gegebenheiten anpassen und einen eignenen HTTP Server passend konfigurieren. Falls Sie damit Probleme haben, dann setzen Sie den Mini Web Server ein, der von Sun für das dynamische Laden von Class Dateien eingesetzt wird. Diesen finden Sie auf der CD und auf dem Server. Dieser Server muss die Class Dateien an den Client liefern.

Das RMI Bootstrap Programm, Client Loader, speziell RMIClientLoader.java, ist der Client, welcher dynamisch Code laden und ausführen kann.

Wie Sie in der Musterlösung sehen, sind die einzelnen Programme in unterschiedlichen Verzeichnissen, also eigentlich auf unterschiedlichen Servern. Der Client lädt also dynamisch seine Aufgaben von einem Server herunter!

Das Faszinierende an diesem Beispiel ist, dass das Bootstrap Programm minimale Informationen über den wahren Client Code enthält. Der Rest wird dynamisch geladen, herunter geladen.

Die einzelnen Programme müssen in folgender Reihenfolge gestartet werden:
1. Bootstrap HTTP Server
2. Bootstrap RMI Server
3. Bootstrap RMI Client

Hinweise zum dynamischen Herunterladen finden Sie im Skript, im "RMI Praxis" Teil.

3.7.1. Voraussetzungen

Sie haben das Hallo Beispiel verstanden. Lesen Sie die entsprechenden Kapitel im RMI Praxis Teil durch.

3.7.2. Programmskelette

```
ClassFileServer.java
ClassServer.java
Hallo.java
HalloImpl.java
MessageObjekt.java
RMIClient.java
RMIClientBootstrapSecurityManager.java (wird nicht benötigt)
RMIClientLoader.java
RMIServer.java
```

3.7.3. Aufgaben

Diese Aufgaben sollten Sie locker lösen können:

1. speichern Sie die Dateien in vier Verzeichnisse:

```
RMIClientLoader
   RMIClientLoader.java
   RMIClientBootstrapSecurityManager.java

RMIClients
   Hallo.java
   HalloImpl.java
   MessageObjekt.java
   RMIClient
   RMIClientBootstrapSecurityManager.java

RMIHTTPServer
   ClassFileServer.java
   ClassServer.java
   RMIClientBootstrapSecurityManager.java

RMIServer
   RMIServer.java
   Hallo.java
   HalloImpl.java
   MessageObjekt.java
   RMIClient
   RMIClientBootstrapSecurityManager.java
```

2. übersetzen Sie die Java Quelldateien des HTTP Servers
3. starten Sie den HTTP Server.
   Passen Sie zuerst Port und Codebase Parameter an.

4. übersetzen Sie alle Java Quelldateien des Clients und Servers;
5. kreieren Sie Stubs (und Skeletons) für die remote Objekt Implementationen.
6. starten Sie das RMI Server Programm
   Die RMI Registry muss nicht gestartet werden: der Server implementiert gleich selbst eine Registry.

7. übersetzen Sie alle Quelldateien des Client Bootstrap Loaders.
8. starten Sie die RMI Registry im Client Bootstapper Verzeichnis.
9. starten Sie den Client Bootstrap Loader.

10. erstellen Sie eine UML und eine JavaDoc Dokumentation. Die JavaDoc Dokumentation für den Klassen Server finden Sie auf der CD oder dem Server.

11. ändern Sie den RMIServer so ab, dass er nach dem Starten eine Verbindung zur URL des Servers aufbaut und eine Datei probehalber liest. Dieser Teil des RMIServers soll eine einfache Ausgabe generieren ("... Verbindung korrekt aufgebaut... Datei konnte gelesen werden... Anzahl Zeichen, die gelesen wurden=...").

Sollten Sie Hilfe benötigen, dann lesen Sie weiter unten weiter.

3.7.4. Musterlösung

Diese befindet sich in den entsprechenden Verzeichnissen unter dem Loesungsverzeichnis: Die Lösung ist aufgeteilt auf die vier obigen Verzeichnisse. In jedem Verzeichnis finden Sie Batch Dateien und Policy Dateien. Wichtig ist, die Policy Datei in das den Verzeichnissen übergelagerte zu kopieren, also ins Loesung- Verzeichnis, da die Programme voll qualifiziert aus dem übergeordneten Verzeichnis gestartet werden.

3.7.5. Demo Applikation

Die Demo Applikation, die Sie erzeugt haben, liefert je nach dem von Ihnen angegebenen Host eine leicht andere Ausgabe.

Im Folgenden sehen Sie die Ausgabe der Musterlösung. IhreLösung müsste eine ähnliche Ausgabe liefern.

Der HTTP Server:

```
HTTP Server fuer Class Dateien Download
CTRL/C zum Abbrechen ...
ClassFileServer: Start
ClassServer: Konstruktor
ClassServer: neuer Listener Thread
ClassFileServer: gestartet; Port=2002;
CLASSPATH=d:\UnterrichtsUnterlagen\ParalleleUndVerteilteSysteme\Kapitel18_R
MInvocation\Uebungen\BootstrapBeispiel\Loesung\RMIHTTPServer
ClassServer: Listen Thread Start
ClassServer: neuer Listener Thread
ClassServer: HTTP Fehlermeldung
ClassServer: finally
ClassServer: Listen Thread Start
ClassServer: neuer Listener Thread
ClassFileServer: Path=RMIClient
ClassFileServer: File Absolute
Path=d:\UnterrichtsUnterlagen\ParalleleUndVerteilteSysteme\Kapitel18_RMInvo
cation\Uebungen\BootstrapBeispiel\Loesung\RMIHTTPServer\RMIClient.class
ClassFileServer: File Laenge=1892
ClassServer: Listen Thread Start
ClassFileServer: Ende Lesen der Dateien
ClassServer: Ausgabe HTTP Header
ClassServer: finally
ClassServer: neuer Listener Thread
ClassFileServer: Path=HalloImpl_Stub
ClassFileServer: File Absolute
Path=d:\UnterrichtsUnterlagen\ParalleleUndVerteilteSysteme\Kapitel18_RMInvo
cation\Uebungen\BootstrapBeispiel\Loesung\RMIHTTPServer\HalloImpl_Stub.clas
s
ClassServer: Listen Thread Start
ClassFileServer: File Laenge=3319
ClassFileServer: Ende Lesen der Dateien
ClassServer: Ausgabe HTTP Header
ClassServer: neuer Listener Thread
ClassFileServer: Path=Hallo
ClassFileServer: File Absolute
Path=d:\UnterrichtsUnterlagen\ParalleleUndVerteilteSysteme\Kapitel18_RMInvo
cation\Uebungen\BootstrapBeispiel\Loesung\RMIHTTPServer\Hallo.class
ClassFileServer: File Laenge=276
```

```
ClassFileServer: Ende Lesen der Dateien
ClassServer: Ausgabe HTTP Header
ClassServer: finally
ClassServer: finally
ClassServer: Listen Thread Start
ClassServer: neuer Listener Thread
ClassFileServer: Path=MessageObjekt
ClassFileServer: File Absolute
Path=d:\UnterrichtsUnterlagen\ParalleleUndVerteilteSysteme\Kapitel18_RMInvo
cation\Uebungen\BootstrapBeispiel\Loesung\RMIHTTPServer\MessageObjekt.class
ClassFileServer: File Laenge=949
ClassServer: Listen Thread Start
ClassFileServer: Ende Lesen der Dateien
ClassServer: Ausgabe HTTP Header
ClassServer: finally
```

Der RMI Server:

```
RMIServer
CTRL/C beendet den Server
RMIServer: Start
RMIServer: RMISecurityManager
RMIServer: Konstruktor
RMIServer: Konstruktor - Kreieren der Registry fuer Port 10009
RMIServer: Registry wurde auf dem Host localhost an Port 10009 kreiert!
RMIServer: Konstruktor - Instanzierung von HalloImpl
RMIServer: Remote Hallo Implementationsobjekt kreiert
RMIServer: Hallo Server ist gebunden!
RMIServer: Abfrage der Registry
RMIServer: urlString=rmi://localhost:10009/Hallo
RMIServer: Laenge=1
           rmi://localhost:10009/Hallo
RMIServer: Binding an Server rmi://localhost:10009/Hallo abgeschlossen;
RMIServer: warten auf Kundenanfragen.
RMIServer: System.getProperties()
RMIServer: Abfrage java.rmi.server.codebase=http://localhost:2002
RMIServer: Instanzierung des RMIServers
MessageObjekt: laufende (Klassen-)Nummer ist  0; Objekt Nummer ist 0
MessageObjekt: laufende (Klassen-)Nummer ist  1; Objekt Nummer ist 1
MessageObjekt: laufende (Klassen-)Nummer ist  2; Objekt Nummer ist 2
MessageObjekt: laufende (Klassen-)Nummer ist  3; Objekt Nummer ist 3
MessageObjekt: laufende (Klassen-)Nummer ist  4; Objekt Nummer ist 4
MessageObjekt: laufende (Klassen-)Nummer ist  5; Objekt Nummer ist 5
MessageObjekt: laufende (Klassen-)Nummer ist  6; Objekt Nummer ist 6
MessageObjekt: laufende (Klassen-)Nummer ist  7; Objekt Nummer ist 7
MessageObjekt: laufende (Klassen-)Nummer ist  8; Objekt Nummer ist 8
MessageObjekt: laufende (Klassen-)Nummer ist  9; Objekt Nummer ist 9
```

Beachten Sie die unterschiedliche Nummerierung zum Client unten! Da eine Kopie des Objekts auf dem Server zum Client gesandt wird, laden beide JVMs ihre eigenen Kopien der Klasse. Die Klassenvariable, als lokale Variable auf dem Server, wird hochgezählt, während diese Variable auf der Kopie auf dem Client konstant bleibt, da jedesmal ein neues Objekt gesandt wird.

Der RMI Client:

```
RMIClient
RMIClientLoader: Start
RMIClientLoader: RMISecurityManager()
RMIClientLoader: Instanzierung des RMIClientLoaders
RMIClientLoader: Konstruktor
RMIClientLoader: System.getProperties()
```

```
RMIClientLoader: Abfrage java.rmi.server.codebase=http://localhost:2002/
RMIClientLoader: URL Konstruktor -
                         java.rmi.server.codebase=http://localhost:2002/
RMIClientLoader: Setzen von java.rmi.server.rminode =
rmi://localhost:10009/
RMIClientLoader: Abfrage der Registry
RMIClientLoader: Anzahl Eintraege=0
RMIClientLoader: Eintraege in der Registry
                 rmi://localhost:10009/Hallo
RMIClientLoader: Anfrage fuer: http://localhost:2002/ und RMIClient
RMIClientLoader: Client Klassen sind geladen
RMIClientLoader: Name der Client Klasse : RMIClient
RMIClientLoader: Client starten - run()
RMIClient: run()
RMIClient: java.rmi.server.rminode= rmi://localhost:10009/
RMIClient: Message vom Server: Hallo, wie geht's!
RMIClient: MessageObjekt: Klassenvariable Nummer 0 Objekt Number is 0
RMIClient: MessageObjekt: Klassenvariable Nummer 0 Objekt Number is 1
RMIClient: MessageObjekt: Klassenvariable Nummer 0 Objekt Number is 2
RMIClient: MessageObjekt: Klassenvariable Nummer 0 Objekt Number is 3
RMIClient: MessageObjekt: Klassenvariable Nummer 0 Objekt Number is 4
RMIClient: MessageObjekt: Klassenvariable Nummer 0 Objekt Number is 5
RMIClient: MessageObjekt: Klassenvariable Nummer 0 Objekt Number is 6
RMIClient: MessageObjekt: Klassenvariable Nummer 0 Objekt Number is 7
RMIClient: MessageObjekt: Klassenvariable Nummer 0 Objekt Number is 8
RMIClient: MessageObjekt: Klassenvariable Nummer 0 Objekt Number is 9
Taste drücken, um fortzusetzen . . .
```

3.7.6. Hilfestellungen

Diese Hilfestellung gibt Ihnen Hinweise auf Lösungen möglicher Probleme. Zuerst mal eine grundsätzliche Bemerkung:
falls Sie Probleme mit dem herunterladen der Klassen haben, dann sollten Sie neben dem Lösen der Aufgabe zum HTTP Server erwägen, einen andern Web Server einzusetzen. Sie können irgend einen Web Server verwenden, müssenaber die Batch Dateien anpassen und vorallem das Root Verzeichnis des Web Servers kennen.

Die folgende Hilfestellung bezieht sich auf die einzelnen Aufgaben.

1. speichern Sie die Dateien gemäss der angegebenen Liste in die entsprechenden Verzeichnisse.

2. starten Sie als erstes den HTTP Server. Er muss die Client und Objekt Class Dateien finden. Sie müssen diese in oder unter das Root Verzeichnis des HTTP Servers kopieren sobald diese erstellt sind.
   Zum Übersetzen können Sie auch die Batch Datei aus der Musterlösung verwenden.

3. starten Sie den HTTP Server
   Sie sollten unbedingt überprüfen, ob der Server die Dateinamen richtig auflöst. Falls dies nicht der Fall ist, sollten Sie
   entweder den Server anpassen
   oder einen andern HTTP Server einsetzen.

   Auf jeden Fall sollten Sie Port und Codebase anpassen.

4. übersetzen Sie alle Dateien des Clients und Servers
   Sollten Sie dabei Probleme haben: die Musterlösung enthält auch dazu Batch Dateien.

5. kopieren Sie die Class Dateien in das dem HTTP Server zugängliche Verzeichnis.

6. generieren Sie Stub und Skeleton für das remote Objekt.
   Dazu steht Ihnen in der Musterlösung eine Batch Datei zur Verfügung. Falls Sie JBuilder verwenden, sollte die Property der Impl Datei bereits so festgelegt sein, dass die Stubs und Skeletons generiert werden.

7. starten Sie den RMI Server
   Sie finden in den Musterlösungen eine Batch Datei, die den Server startet. Passen Sie diese Datei Ihren Gegebenheiten (Verzeichnisstrukturen) an.

8. übersetzen Sie alle Quelldateien des Bootstrap Loader Programms
   Die entsprechende Batch Datei finden Sie im entsprechenden Verzeichnis der Musterlösung.

9. Security Manager
   Den RMIClientBootstrapSecurityManager benötigen Sie nicht mehr, da Java 2 die Zugriffsprobleme auf andere Art und Weise gelöst hat. Sie benötigen aber eine Policy Datei.

10. starten Sie den Client
    Sie finden die entsprechende Batch Datei im entsprechenden Verzeichnis.

3.8. Distributed Garbage Collector

In dieser Übung sind Sie überwiegend Zuschauer. Sie müssen lediglich den Programmcode anpassen und die Programme starten, sowie wie immer, die UML Dokumentation erstellen.

Die Anpassungen, die Sie vornehmen müssen beziehen sich auf den Host, `HOST_NAME` und eventuell einen Port. Vermutlich müssen Sie sich den Programmcode zu Gemüte führen, um zu verstehen, was genau passiert, wie die Programme ablaufen.

In dieser Übung arbeiten Sie mit zwei remote Objekten, Hallo und MessageObjekt. Beide Objekte sind in der Lage Informationen über die Objekte und deren Lebenszyklus auszugeben. Dies geschieht mit Hilfe der beiden Methoden `finalize`() und `unreferenced`().

Ein remote Objekt kann das `Unreferenced` Interface und seine einzige Methode `unreferenced`() implementieren. Diese Methode wird vom DGC aufgerufen, falls dieser die letzte remote Referenz auf ein Objekt entfernt. `MessageObjektImpl` und `HalloImpl` geben jeweils eine Meldung aus, falls sie aufgerufen werden.

MessageObjektImpl und HalloImpl implementieren auch noch die finalize() Methode. Diese wird aufgerufen, falls der lokale Garbage Collector ein Objekt zerstören will und den Speicherplatz frei geben will. Die Objekte geben eine kurze Meldung aus, falls diese Methoden aufgerufen werden.

Die Theorie zu dieser Übung finden Sie im "RMI Praxis" Teil im Abschnitt über Distributed Garbage Collector.

Für diese Übung benötigen Sie einen RMI Server und mehrere Clients. Sie können den Client einfach mehrfach starten, um etwas mehr Aktivitäten auf dem Server zu produzieren.

Damit Sie ein besseres Verständnis über DGC bekommen, sollten Sie mit einem Parameter arbeiten und unterschiedliche Kombinationen testen:

```
java -Djava.rmi.dgc.leaseValue=10000 RMIServer
```

Die Variable `leaseValue` wird in Millisekunden angeben. Je nach Wert dieser Variable muss der DGC länger oder weniger lange warten bevor das Objekt aus dem Heap entfernt werden darf.

Die zweite Variable, mit der Sie experimentieren sollten, ist die Grösse des Java Heaps. Diese wird über das Flag `mx` auf der Kommandozeile gesteuert.

`-Xmx`*n* spezifiziert die maximale Grösse des Speicherallokationspools in Bytes. Der Wert muss ein Vielfaches von 1024 und grösser als zwei MB sein. Falls Sie die Grösse in MB angeben wollen, müssen Sie ein M anhängen. Der Standardwert ist 64 MB.

Beispiele:

```
-Xmx83886080
-Xmx81920k
-Xmx80m
```

3.8.1. Voraussetzungen

Sie haben das Hallo Beispiel verstanden. Lesen Sie die entsprechenden Kapitel im RMI Praxis Teil durch. Dieses Beispiel verwendet im wesentlichen die gleiche Logik, erweitert aber die Programme durch finalize und unreferenced.

Beide Methoden zeigen den Status der Objekte auf der Zeitachse, speziell das Löschen der Objekte, das Entfernen aus dem Speicher, dem Heap.

3.8.2. Programmskelette

```
Hallo.java
HalloImpl.java
MessageObjekt.java
MessageObjektImpl.java
RMIClient.java
RMIServer.java
```

3.8.3. Aufgaben

Erstellen Sie ein Projekt mit allen Java Quellen. Im Quellcode müssn Sie verschiedene Änderungen durchführen.

1. ändern Sie im RMI Client den HOST_NAME so, dass er Ihrer Konfigurationentspricht. Sie können auch eine Version bauen, bei der der Hostname als Parameter auf der Kommandozeile eingegeben wird.

2. übersetzen Sie alle Java Quelldateien..
   Die Klassendateien `Hallo, HalloImpl, MessageObjekt, MessageObjektImpl, RMIClient` und `RMIServer.`

3. starten Sie den RMI Server mit dem kleinstmöglichen Heap.

4. starten Sie zwei oder mehr Instanzen des Clients.

5. Erstellen Sie eine UML Dokumentation des Projekts.
   Wie sieht das Sequenzdiagramm aus?

Beobachten Sie speziell das Server Fenster. Die Clients sind eher uninteressant. Sie sollten in den Client Fenstern sehen, dass jeweils unterschiedliche Objekte kreiert werden. Im Server Fenster erkennen Sie, wann ein Objekt seine Leasingdauer verbraucht hat und frei gegeben wird. Dies geschieht in den zwei Stufen:

1. unreferenced
   Jetzt wird das Objekt in eine Liste eingetragen, die vom Garbage Collector und vom DGC gelesen wird. Darin stehen Kandidaten von löschbaren Objekten.

2. finalize
   Damit hat die letzte Stunde, die letzte Millisekunde des Objekts geschlagen.

Beobachten Sie das Server Fenster auch noch eine Weile nachdem der Client bereits gestoppt wurde. Sie sehen dann sehr schön, wie der Server nach kurzer Zeit weiter aufräumt und alle Objekte löscht.

3.8.4. Musterlösung

```
Hallo.java
HalloImpl.java
MessageObjekt.java
MessageObjektImpl.java
RMIServer.java
RMIClient.java
```

3.8.5. Demo Applikation

Die Demo Applikation, die Sie erzeugt haben, liefert je nach dem von Ihnen angegebenen Host eine leicht andere Ausgabe.

Im Folgenden sehen Sie die Ausgabe der Musterlösung. IhreLösung müsste eine ähnliche Ausgabe liefern.

Der RMIServer:
Direkt nach dem Starten liefert der Server folgende Ausgabe:

```
RMIServer
CTRL/C beendet den Server
Registry wurde auf dem Host Computer localhost an Port 10007 exportiert
Remote Hallo Service Implementationsobjekt kreiert
Binden abgeschlossen, warten auf Client Anfragen.
```

Nach dem Starten des Clients meldet sich der Server:

```
MessageObjekt: Klassen-Nummer ist #0 Objekt-Nummer ist #0
MessageObjekt: Klassen-Nummer ist #1 Objekt-Nummer ist #1
MessageObjekt: Klassen-Nummer ist #2 Objekt-Nummer ist #2
MessageObjekt: Klassen-Nummer ist #3 Objekt-Nummer ist #3
MessageObjekt: Klassen-Nummer ist #4 Objekt-Nummer ist #4
MessageObjekt: Klassen-Nummer ist #5 Objekt-Nummer ist #5
MessageObjekt: Klassen-Nummer ist #6 Objekt-Nummer ist #6
MessageObjekt: Klassen-Nummer ist #7 Objekt-Nummer ist #7
MessageObjekt: Klassen-Nummer ist #8 Objekt-Nummer ist #8
MessageObjekt: Klassen-Nummer ist #9 Objekt-Nummer ist #9
MessageObjekt: Klassen-Nummer ist #10 Objekt-Nummer ist #10
MessageObjekt: Klassen-Nummer ist #11 Objekt-Nummer ist #11
MessageObjekt: unreferenced() fuer Objekt #: 4
MessageObjekt: unreferenced() fuer Objekt #: 0
MessageObjekt: unreferenced() fuer Objekt #: 1
MessageObjekt: unreferenced() fuer Objekt #: 2
MessageObjekt: unreferenced() fuer Objekt #: 3
MessageObjekt: unreferenced() fuer Objekt #: 5
MessageObjekt: unreferenced() fuer Objekt #: 6
MessageObjekt: Klassen-Nummer ist #12 Objekt-Nummer ist #12
MessageObjekt: Klassen-Nummer ist #13 Objekt-Nummer ist #13
MessageObjekt: Klassen-Nummer ist #14 Objekt-Nummer ist #14
MessageObjekt: Klassen-Nummer ist #15 Objekt-Nummer ist #15
MessageObjekt: Klassen-Nummer ist #16 Objekt-Nummer ist #16
MessageObjekt: Klassen-Nummer ist #17 Objekt-Nummer ist #17
MessageObjekt: Finalize fuer Objekt #: 6
MessageObjekt: Finalize fuer Objekt #: 5
MessageObjekt: Finalize fuer Objekt #: 4
MessageObjekt: Finalize fuer Objekt #: 3
MessageObjekt: Finalize fuer Objekt #: 2
```

```
MessageObjekt: Finalize fuer Objekt #: 1
MessageObjekt: Finalize fuer Objekt #: 0
...
```

Der RMIClient:
Direkt nach dem Starten liefert der erste Client folgende Ausgabe:

```
RMIClient
HalloService lookup erfolgreich
Message vom Server: Hallo Du da!
MessageObjekt: Klasse-Nummer ist #1 Objekt Nummer ist #1
MessageObjekt: Klasse-Nummer ist #2 Objekt Nummer ist #2
MessageObjekt: Klasse-Nummer ist #3 Objekt Nummer ist #3
MessageObjekt: Klasse-Nummer ist #4 Objekt Nummer ist #4
MessageObjekt: Klasse-Nummer ist #5 Objekt Nummer ist #5
MessageObjekt: Klasse-Nummer ist #6 Objekt Nummer ist #6
MessageObjekt: Klasse-Nummer ist #7 Objekt Nummer ist #7
MessageObjekt: Klasse-Nummer ist #8 Objekt Nummer ist #8
MessageObjekt: Klasse-Nummer ist #9 Objekt Nummer ist #9
MessageObjekt: Klasse-Nummer ist #10 Objekt Nummer ist #10
MessageObjekt: Klasse-Nummer ist #11 Objekt Nummer ist #11
MessageObjekt: Klasse-Nummer ist #12 Objekt Nummer ist #12
MessageObjekt: Klasse-Nummer ist #13 Objekt Nummer ist #13
MessageObjekt: Klasse-Nummer ist #14 Objekt Nummer ist #14
MessageObjekt: Klasse-Nummer ist #15 Objekt Nummer ist #15
...
```

Der zweite und die weiteren Clients liefern entsprechende Ausgaben.

3.8.6. Hilfestellungen

Zu jeder Aufgabe wurden spezielle Batch Dateien kreiert. Falls Sie Probleme haben, dürften diese Dateien die meisten dieser Probleme lösen.

Spezielle Hinweise zu den einzelnen Aufgaben:

1. ändern Sie den Hostnamen und ändern Sie das Programm so, dass es den Hostnamen ab Kommandozeile liest. Dies haben Sie bereits so oft getan, dass es für Sie kein Problem sein dürfte.

2. das Übersetzen der Quelldateien ist denkbar einfach.
   Auch hierzu steht Ihnen eine Batch Datei zur Verfügung.

3. zum Starten des RMI Servers wurde eine Batch Datei erstellt. Es liegt an Ihnen die Heapgrösse zu testen und insbesondere auch Performanceunterschiede zu notieren.

4. starten Sie einen oder mehrere Clients
   Dazu steht Ihnen eine Batch Datei zur Verfügung. Sie sehen das Verhalten des DGC auch mit einem einzelnen Client. Aber ein echt verteiltes System sollte komplexer sein.

5. warten Sie nach dem Ende aller Clients einige Minuten und beobachten Sie das Server Fenster. Sie werden sehen, wie die einzelnen Objekte nach Ablauf der Leasingzeit schrittweise verschwinden.

3.9. Serialisierung von Remote Objekten - Serverseite

Diese Übung ist eine Anwendung des Delegatin Entwurfmuster. Die Theorie dazu finden Sie im Abschnitt Serialisierung von Remote Objekten im "RMI Praxis" Teil.

3.9.1. Voraussetzungen

Sie müssen das entsprechende Kapitel im Theorieteil durchgearbeitet haben und das Hallo Beispiel bestens kennen.

3.9.2. Programmskelette

```
RemoteModelMgr.java
RemoteModelMgrImpl.java
RemoteModelImpl.java
LocalRemoteServer.java
LocalModel.java
RemoteModelRef.java
```

3.9.3. Aufgaben

Die einzelnen Aufgaben bestehen primär aus dem zum Laufen bringen bestehender Applikationen. Lesen Sie die Lernziele, weiter oben und überlegen Sie sich, ob Sie diese erreicht haben.

1. übersetzen Sie alle Java Quelldateien
   Dafür steht Ihnen in der Musterlösung eine Batch Datei zur Verfügung.

2. kreieren Sie Stubs (und Skeletons) für die remote Objekte
   Sie können dies entweder auf Batch Ebene oder im JBuilder machen:
   im JBuilder müssen Sie die Datei im Projekt-Fenster anwählen und die rechte Maustaste drücken. Unter Properties können Sie dann die Generierung von Stubs anwählen.

3. starten Sie den Server

3.9.4. Musterlösung

Diese befindet sich im Verzeichnis Loesung und enthält auch alle benötigten Batch Dateien.

```
RemoteModelMgr.java
RemoteModelMgrImpl.java
RemoteModelImpl.java
LocalRemoteServer.java
LocalModel.java
RemoteModelRef.java
```

3.9.5. Demo Applikation

Als einzige Demo Applikation werden Sie den Server starten können. Dieser benötigt keine Registry, weil der Server eine eigene Instanz der Registry generiert.

```
@echo off
echo RMIServer
echo CTRL/C beendet den Server
java -Djava.security.policy=java.policy LocalRemoteServer localhost
pause
```

Nach dem Starten sehen Sie folgende Ausgaben auf dem Server:

LocalRemoteServer:

```
RMIServer
CTRL/C beendet den Server
Registry wurde auf dem Host localhost an Port 10009 kreiert.
RemoteModelImpl Objekt kreiert.
RemoteModel wurde gebunden.
Warten auf Kundenanfragen ...
```

3.9.6. Hilfestellungen

Die Aufgaben werden in der Musterlösungen mit Batch Dateien versehen, damit Sie nötigenfalls mit etwas starten können.

Hilfestellungen zu den einzelnen Aufgaben:

1. übersetzen aller Java Quellen
   Sie finden dazu die übliche Batch Datei.

2. Generieren Sie Stubs (und nötigenfalls Skeletons):
   In JBuilder durch anwählen der entsprechenden Properties, wie in der vorigen Übung beschrieben. Beachten Sie, dass dabei viele veraltete Teile mit generiert werden: Skeletons sind nicht mehr nötig.
   Versuchen Sie die Impl Dateien mit dem RMI Compiler zu generieren.

3. starten Sie nun den Server

3.10. Serialisierung von Remote Objekten - Clientseite

In dieser Übung schreiben Sie nun noch den Client zum Server. Da Sie sicher schon müde sind, nehme ich Ihnen dies wieder weitestgehend ab.

Sie sollten aber erkennen, inwiefern das Delegation Pattern implementiert wird.

3.10.1. Voraussetzungen

Lesen Sie die entsprechende Theorie im "RMI Praxis" Teil und zusätzlich das Kapitel über das Delegation Pattern im GoF Buch.

3.10.2. Programmskelette

`LocalRemoteClient.java`

3.10.3. Aufgaben

Die Aufgaben bestehenprimär im gewinnen des Verständnisses!

1. Lesen Sie die Theorie im "RMI Praxis" Teil.
   Der entsprechende Abschnitt sollte Ihnen Hintergrundinformationen zum Thema liefern.

2. Lesen Sie die Beschreibung des Delegation Pattern im GoF Buch nach.
   Besprechen Sie den Text mit Ihren Kollegen!

3. Kopieren Sie Stubs aus der Server Übung in Ihr Entwicklungsverzeichnis.

4. übersetzen Sie das Client Programm.
   Damit keine Fehler auftreten müssen Sie einige Quellen aus der Serveraufgabe in das Client Verzeichnis kopieren.

5. starten Sie den RMI Compiler zur Generierung der Stubs.
   Falls Sie dies im JBuilder tun, wie in der vorigen Übungen beschrieben, werden Sie einige Warnungen erhalten, da ab Java 2 keine Skeletons mehr benötigt werden.

6. dokumentieren Sie die Programme mit JavaDoc:
   ergänzen Sie den Source Code durch JavaDoc Direktiven und generieren Sie die JavaDoc mit dem JavaDoc Werkzeug.

7. dokumentieren Sie das Delegations Pattern, so wie es hier eingesetzt wird, mit Hilfe von UML. Besprechen Sie die unterschiedliche Darstellung in UML und im GoF Buch.

3.10.4. Musterlösung

`LocalRemoteClient.java`

3.10.5. Demo Applikation

Nach dem Starten des Servers aus der vorhergehenden Übung sollten Sie den Client problemlos starten können, sofern Sie auch noch eine Policy für den Security Manager angeben.

Sie finden auf dem Server, der CD, eine Musterlösung, welche alle Batch Dateien enthält, die zum testen des Programms benötigt werden.

Hier zur Wiederholung die Ausgabe der Servers:
LocalRemoteServer:

```
RMIServer
CTRL/C beendet den Server
Registry wurde auf dem Host localhost an Port 10009 kreiert.
RemoteModelImpl Objekt kreiert.
RemoteModel wurde gebunden.
Warten auf Kundenanfragen ...
RemoteModelImpl...Konstruktor abgeschlossen
LokalesModell... Version: Version 1.0ax.2b4
RemoteModelImpl...Versions Nummer: Version 1.0ax.2b4
LokalesModell... Version: Version 1.0ax.2b4
RemoteModelImpl...Versions Nummer: Version 1.0ax.2b4
LokalesModell... Version: Version 1.0ax.2b4
RemoteModelImpl...Versions Nummer: Version 1.0ax.2b4
```

LocalRemoteClient:

```
RMIClient
RemoteModelManager Lookup erfolgreich
Remote Version: Version 1.0ax.2b4
Lokale Version des geladenen Modells
LokalesModell... Version: Version 1.0ax.2b4
Lokale Version: Version 1.0ax.2b4
Taste drücken, um fortzusetzen . . .
```

3.10.6. Hilfestellungen

Sie sollten erkennen, wie das Delegation Pattern funktioniert und wie es hier eingesetzt wird. Die Beschreibung im GoF Buch ist etwas umständlich. Aber es gibt keine Alternative zu diesem Buch. Viele Bücher über Design Patterns zeigen, dass die Autoren die Konzepte nicht verstanden haben. Sie müssen sich also durch das Buch bzw. diese Pattern Beschreibung durcharbeiten.

Der praktische Teil ist einfach, sofern Sie nicht vergessen, die Quelldateien aus der vorherigen Übung in Ihr Arbeitsverzeichnis zu kopieren.

Zu beachten ist auch noch, dass der Client `LocalRemoteClient` ist. Beim Starten müssen Sie also:

```
java LocalRemoteClient
```

eingeben.

Damit schliessen wir die Übungen zum Thema RMI ab.