In diesem Kursteil

- Modul 2 : Remote Method Invocation RMI
  - Modul Einleitung
  - Was ist Java RMI?
  - RMI Architektur Übersicht
  - Der Transport Layer
  - Garbage Collection
  - Remote Reference Layer
  - RMI Stubs und Skeletons
  - RMI Packages und Hierarchien
  - Kreieren einer RMI Applikation
  - RMI Security
  - Remote Methode Invocation Praktische Übung
  - Quiz
  - Zusammenfassung

Java in Verteilte Systeme

1.1. Modul 1 : Remote Method Invocation (RMI)

1.1.1. Einleitung

Das Remote Method Invocation (RMI) API gestatte es dem Java Entwickler Programme zu schreiben, welche auf remote Objekte zugreifen, genauso wie wenn diese Objekte lokal wären.

Analog zu den sogenannten Remote Procedure Calls (RPC) abstrahiert RMI die Socket Verbindung und die Datenumwandlungen, welche für eine Kommunikation mit einem entfernten Rechner benötigt werden. Dadurch werden entfernte Methodenaufrufe genau so gemacht, wie lokale Methodenaufrufe.

1.1.1.1. Lernziele

Nach dem Durcharbeiten dieser Unterlagen sollten Sie in der Lage sein:

- die RMI Architektur zu beschreiben, inklusive seinen verschiedenen Layern und dem (Distributed) Garbage Collector.
- RMI Server und Clients in Java zu implementieren.
- Client Stuby und Skeletons für remote Services mit Hilfe des Stub Compilers zu generieren.
- die Funktionsweise der RMI Registry zu beschreiben.
- eine RMI Applikation zu entwickeln, um eine bestimmte verteilte Aufgabe zu lösen.

1.1.1.2. Referenzen

Teile dieses Moduls stammen teilweise oder ganz aus

- "The Java Remote Method Invocation Specification"
  `http://java.sun.com/products/javaspaces/index.html`
- "Java RMI Tutorial"
  `http://java.sun.com/products/javaspaces/index.html`
- "Frequently Asked Questions, RMI and Object Serialization"
  `http://java.sun.com/products/javaspaces/index.html`

1.1.2. Was ist Java RMI?

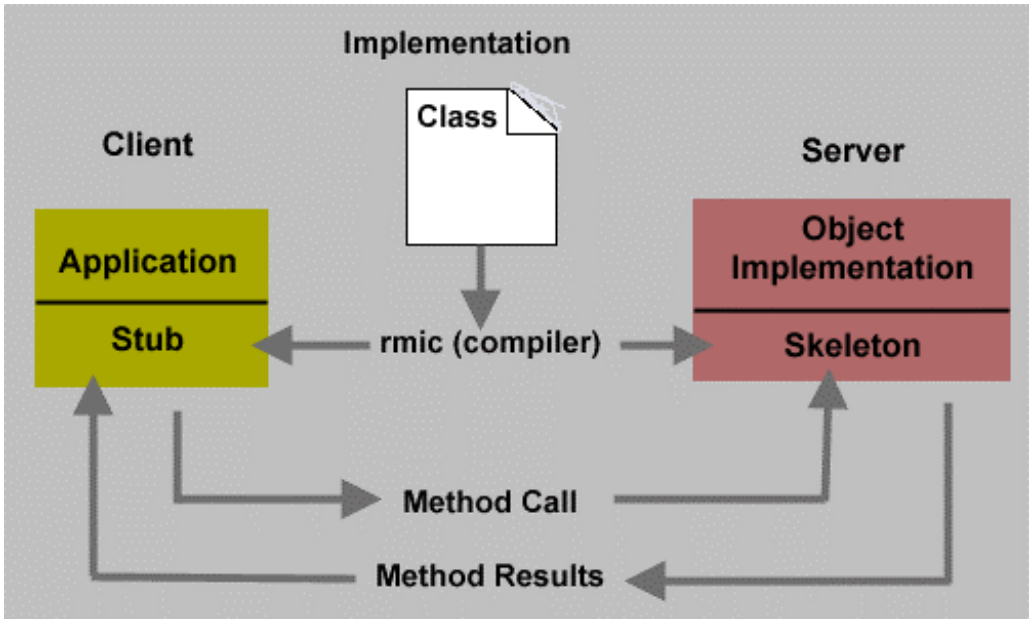
Das RMI API besteht aus einem Set von Klassen und Interfaces, mit deren Hilfe der Entwickler Methoden entfernter Objekte aufrufen kann, also Methoden von Objekten, die sich zur Laufzeit in einer anderen Java Virtuellen Maschine befinden. Diese "remote" oder "Server" JVM kann sich dabei auf der selben oder auf unterschiedlichen Maschinen befinden als der RMI Client. Java RMI ist im Gegensatz zu RPC eine reine Java Lösung.

In diesem Modul lernen Sie, wie man RMI Interfaces schreibt, die vom Clinet Code benutzt werden. und wie RMI Implementationsklassen aussehen, mit denen Objekte auf dem Server instanziert werden. Mit Hilfe der RMI Interface Beschreibungen werden Stubs (Client seitiger Code) und Skeletons (Server seitiger Code) mit Hilfe des `rmic`, des RMI Compilers generiert.

In verteilten Systemen sind ausgefeilte Ausfall- und Recovery- Mechanismen für verteilte Programme sehr wesentlich, einfach weil es viel mehr Ausfallmodi gibt. In diesem Modul lernen Sie einige dieser Java RMI definierten Exception Typen kennen, zusammen mit Hinweisen, wie Sie diese in Ihren Programmen einsetzen sollten.

1.1.3. Übersicht über die RMI Architektur

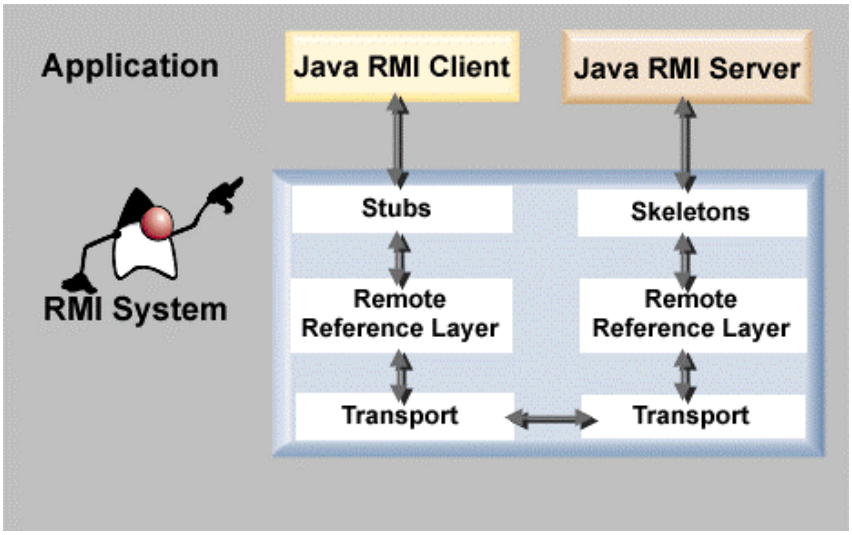
Der Aufruf einer remote Methode durch einen Client auf einem Server wird mit Hilfe verschiedener Layer des RMI Systems auf der Client Seite zum Transport Layer geleitet, dann zum Server transportiert und schliesslich mehrere Layer nach oben zum Server Objekt weitergeleitet.



Schematisch ist dies im obigen Bild (aus der RMI Dokumentation) wiedergegeben.
Das Ganze klingt trotzdem sehr komplex. Wie geschieht den dieser Transport und welche Layer gibt es?

Als Entwickler sind Sie lediglich für die Definition der Interfaces und der Implementationsklassen und das Generieren der Stub und Skeleton Klassen (mit Hilfe von `rmic`) verantwortlich.

Der Remore Reference Layer (RRL) und der Transport Layer kümmern sich um den Rest. Diese Layer könnten sich sogar von Release zu Release verändern, ohne dass Ihre Anwendungsprogramme davon betroffen wären.



Stubs und Skeleton, die Java Programme, die auf Grund unserer Interface Definitionen kreiert werden (wobei Skeletons bei neueren Releases nicht mehr benötigt werden), sind sozusagen die Schnittstelle zum Anwendungsprogramm und dem Reference Layer.

Der Reference Layer ist nötig, da Referenzen in der einen JVM auf Referenzen in einer eventuell entfernten JVM abgebildet werden müssen.

1.1.4. Der Transport Layer

Schauen wir uns als erstes die Funktionsweise des Transport Layers an.

Der Transport Layer ist für den Verbindungsaufbau, das Managementd der Verbindung und das Verfolgen und Verwalten der remote Objekte (den Zielobjekten der remote Calls) im Adressraum verantwortlich.

Der Transport Layer hat folgende Aufgaben:

- er empfängt eine Anfrage vom Remote Reference Layer auf der Clientseite.
- er lokalisiert den RMI Server, welcher das verlangte remote Objekt enthält.
- er baut eine Socket Verbindung zu diesem Server auf.
- er liefert diese Verbindung / Verbindungsinformation an den clientseitigen Remote Reference Layer.
- er trägt das remote Objekt in eine Tabelle ein, in der remote Objekte stehen, mit denen kommuniziert werden kann.
- er überwacht die "liveness" dieser Verbindung. Der Client selbst kann eine Verbindung zu einem RMI Server nicht selbst abbrechen. Dies ist eine der Aufgaben des Transport Layers.

1.1.4.1. Socket Verbindungen

Zur Zeit unterstützt Java RMI lediglich TCP Sockets. RMI benutzt typischerweise zwei Socket Verbindung: eine für die Methodenaufrufe und eine für den Distributed Garbage Collection (DGC). RMI wird versuchen, bestehende Socket Verbindungen mehrfach zu nutzen, für mehrere Objekte vom selben Server. Falls jedoch eine Socket Verbindung besetzt ist, wird automatisch eine neue Socket Verbindung aufgebaut.

Ein remote Objekt besteht aus einem Server Endpunkt und einem Objektidentifier. Dies wird als *live reference* bezeichnet.

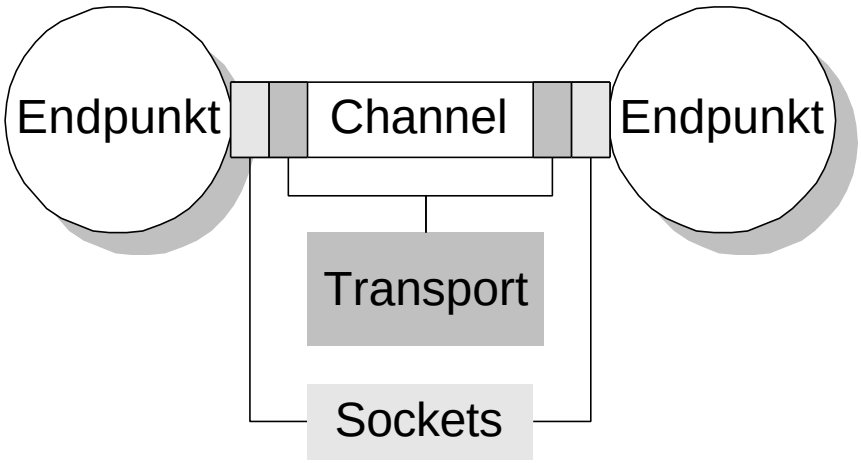
Bei gegebener live reference kann der Transport die Endpunkt Information benutzen, um eine Verbindung zum Adressraum aufzubauen, in dem das remote Objekt vorhanden ist.
Auf der Serverseite kann der Transport den Objektidentifier benutzen, um das Taregt des remote calls zu finden. Der Transport Layer für das RMI System besteht aus folgenden vier Abstraktionen:

- die **Verbindung** (*connection*)wird benutzt, um Daten zu transferieren (Eingabe/Ausgabe).
  Für jede Verbindung existiert ein Kommunikations- Kanel (*channel*), mindestens zwei Endpunkte, plus ein **Transport** (*transport*).

- ein **Endpunkt** (*endpoint*) beschreibt einen Adressraum oder Java Virtual Machine. Ein Endpunkt kann auf seinen Transport abgebildet werden. Bei gegebenem Endpunkt kann also eine spezifische Transportinstanz bestimmt werden.

- ein Channel wird als virtuelle Verbindung zwischen zwei Adressräumen eingesetzt. Der Kanal ist für das Management der Verbindungen zwischen dem lokalen Adressraum und dem remote Adressraum verantwortlich.

- ein **Transport** ist für das Management eines spezifischen Kanals zuständig.
  Der Kanal definiert, wie die konkrete Darstellung des Endpunktes aussieht. Pro Adressraumpaar oder Paar von Endpunkten besteht genau ein Transport. Bei gegebenem Endpunkt zu einem entfernten Adressraum baut ein Transport einen Kanal zwischen sich selbst und diesem Adressraum auf. Der Transport ist auch für die Annahme von ankommenden Anfragen an diesen Adressraum zuständig. Dabei wird ein Verbindungsobjekt für diese Anfrage kreiert und mit den höheren Layern kommuniziert.



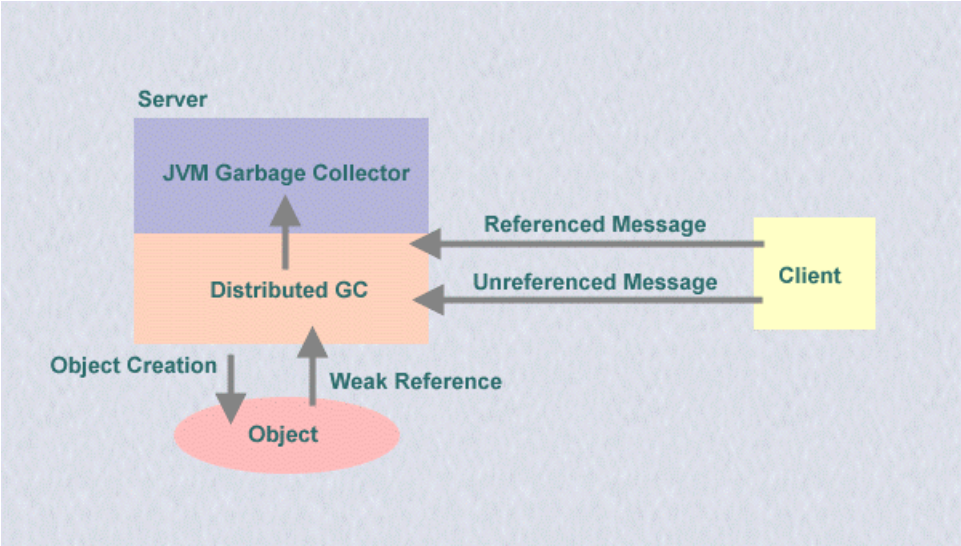
1.1.5. Garbage Collection

RMI benutzt ein Referenz Counting Garbage Collection. Alle live references innerhalb jeder JVM werden dabei berücksichtigt. Immer wenn eine live reference in eine JVM eintritt, wird deren Referenzzähler um eins erhöht. Falls ein Objekt keine Referenzen mehr besitzt, reduziert die Objekt Finalisation (*finalizer*) den Referenzzähler. Falls alle Referenzen aufgelöst wurden, wird dem Server eine "unreferenced" Meldung an den Server gesandt.

Falls ein remote Objekt von keinem Client mehr referenziert wird, wird dies als '*weak reference*' bezeichnet. Eine weak reference gestattet es dem Server Garbage Collector das Objekt zu löschen, sofern keine andere (lokale) Referenz auf dieses Objekt existiert. Solange lokale Referenzen existieren, kann das remote Objekt nicht als Abfall betrachtet werden. Es könnte nach als Parameter oder als Rückgabe eines remote calls eingesetzt werden.

1.1.5.1. Der Garbage Collection Prozess

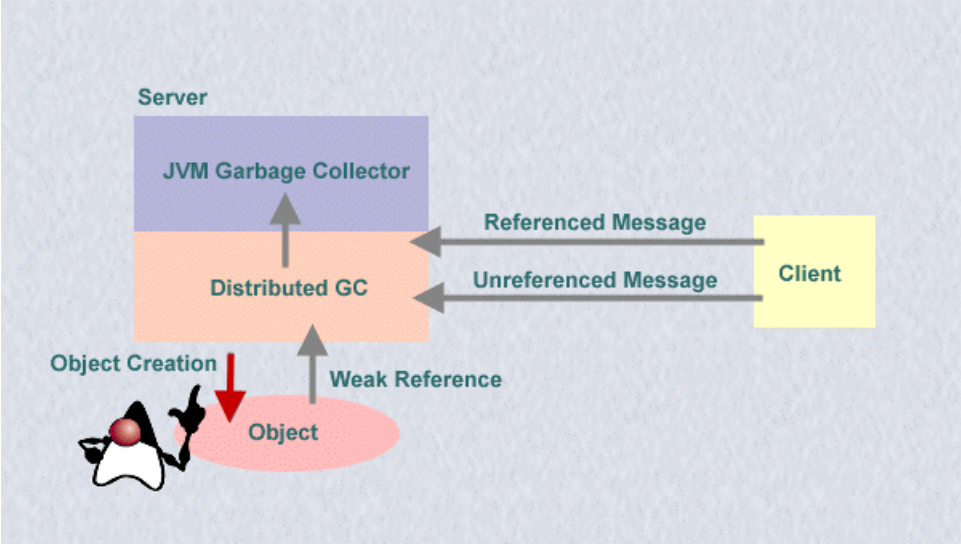
Die folgende Abbildung illustriert den fünfstufigen *Distributed Garbage Collecton* Prozess.



1) Kreieren und starten eines Objekts durch den Server oder die Implementation.
2) Die remote Referenz etabliert eine '*weak reference*' zum Objekt.
3) Falls der Client dieses Objekt verlangt, kreiert die Client JVM eine '*live reference*' und die erste Referenz auf dieses Objekt sendet eine '*Referenced*' Message zum Server.
4) Sobald das Objekt auf dem Client nicht mehr benötigt wird, sendet der Client eine '*Unreferenced*' Message an den Server
5) Falls der Referenzzähler auf dem Objekt auf Null zurück geht und es keine lokalen Referenzen gibt, kann die Objektreferenz dem lokalen GC (garbage collector) gemeldet werden.
   Der DGC kümmert sich darum; er ist Teil des RMI Laufzeitsystems, speziell des Transport Layers.

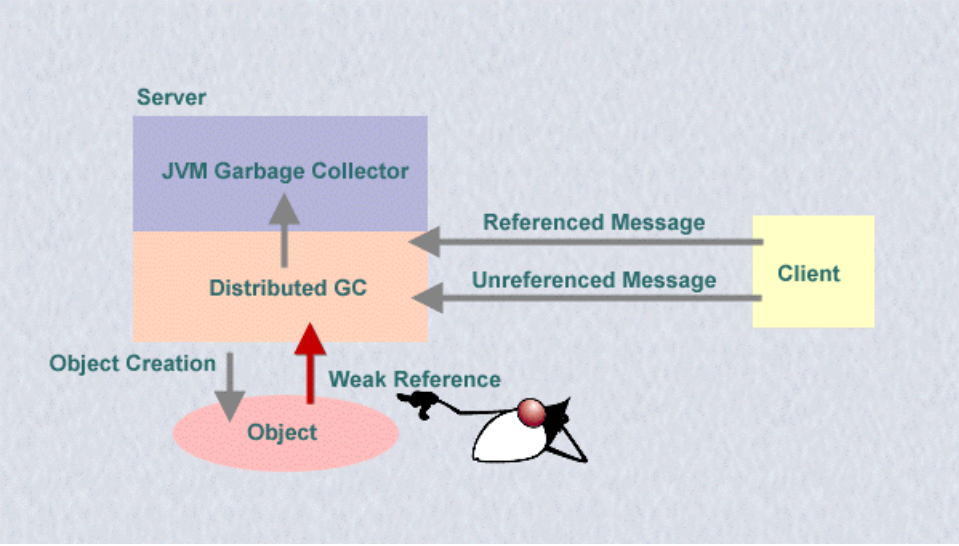
1.1.5.1.1. Distributed Garbage Collection - Schritt 1

Kreieren und starten eines Objekts durch den Server oder die Implementation.



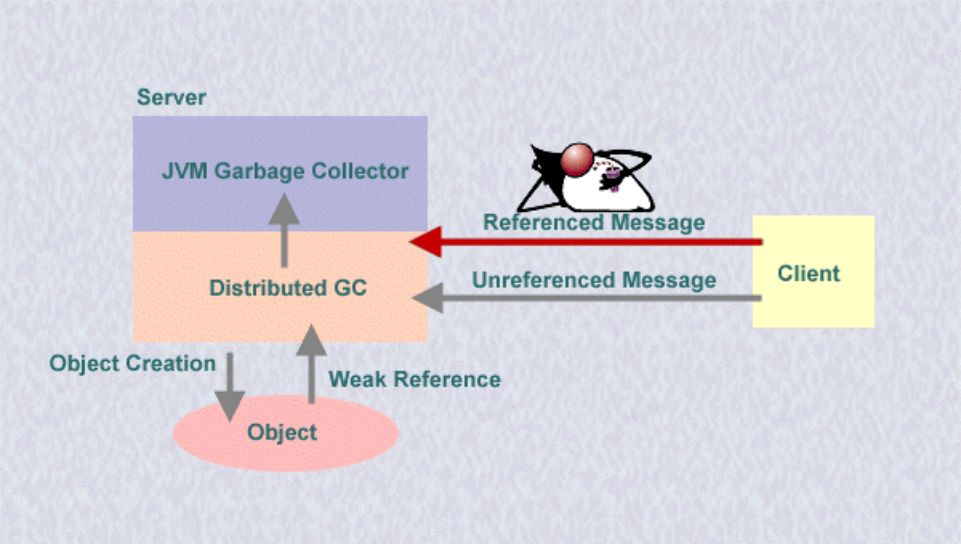
1.1.5.1.2. Distributed Garbage Collection - Schritt 2

Die remote Referenz etabliert eine *'weak reference*' zum Objekt.



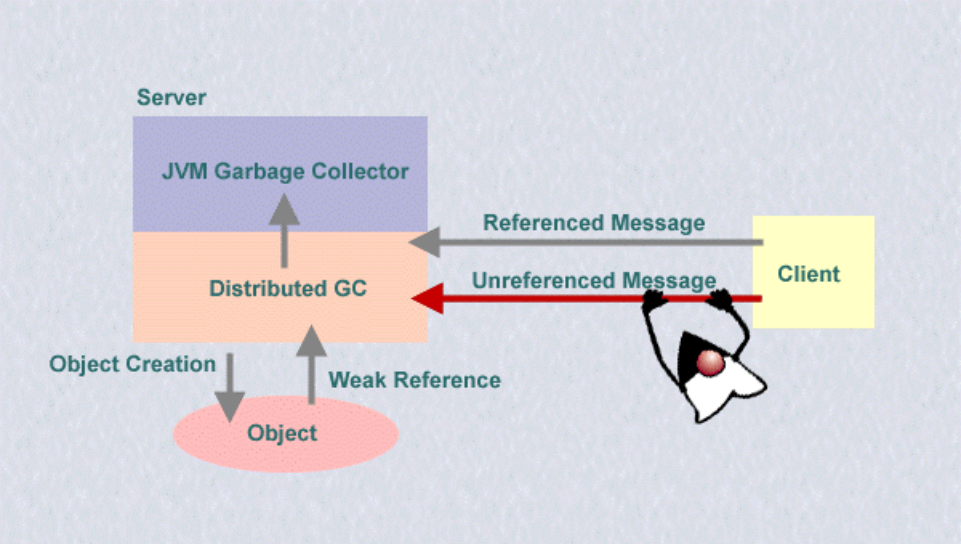
1.1.5.1.3. Distributed Garbage Collection - Schritt 3

Falls der Client dieses Objekt verlangt, kreiert die Client JVM eine '*live reference*' und die erste Referenz auf dieses Objekt sendet eine '*Referenced*' Message zum Server



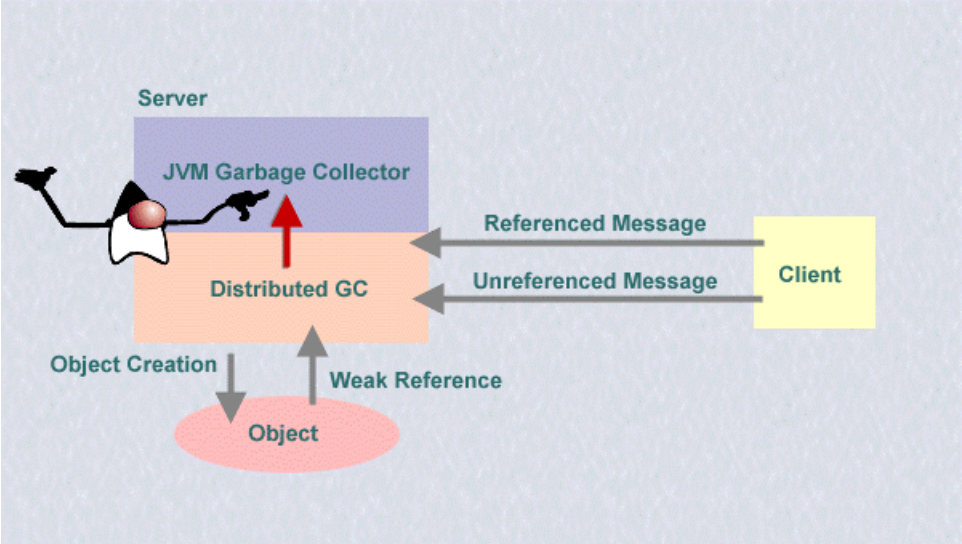
1.1.5.1.4. Distributed Garbage Collection - Schritt 4

Sobald das Objekt auf dem Client nicht mehr benötigt wird, sendet der Client eine '*Unreferenced*' Message an den Server



1.1.5.1.5. Distributed Garbage Collection - Schritt 4

Falls der Referenzzähler auf dem Objekt auf Null zurück geht und es keine lokalen Referenzen gibt, kann die Objektreferenz dem lokalen GC (garbage collector) gemeldet werden



1.1.6. Remote Reference Layer

Der Remote Reference Layer (RRL) ist für die korrekte Semantik der Methodenaufrufe zuständig. Dieser Layer kommuniziert zwischen den Stubs / Skeletons und dem tiefer liegenden Transport Interface. Dazu wird ein spezielles Remote Reference Protokoll verwendet, welches unabhängig ist von Client Stub und Server Skeletons. Zu den Aufgaben des RRL's gehört auch das Verwalten der Referenzen auf remote Objekte und Wiederverbindungsaufbau, falls ein Objekt nicht mehr verfügbar sein sollte.

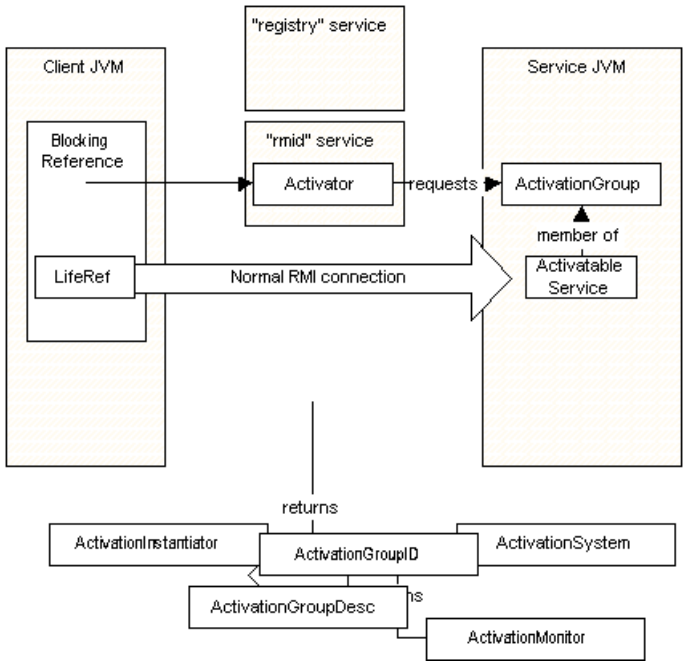
RRL besitzt zwei kooperierende Komponenten: die Client Seite und die Server Seite.

- Die clientseitige Komponente enthält spezifische Informationen über den remote Server und kommuniziert mit Hilfe des Transport Layers mit der serverseitigen Komponente.
- Die serverseitige Komponente implementiert die spezifische Referenzsemantik bevor ein remote Methodenaufruf an das Skeleton übergeben wird.

Die Referenzsemantik für den Server werden auch durch den RRL abgehandelt. RRL abstrahiert die unterschiedlichen Arten, auf die ein Objekt referenziert wird, welches

- auf Servern implementiert sind, welche dauernd auf einer Maschine laufen.
- auf Servern laufen, welche nur dann aktiviert werden, falls eine remote Methode auf ihnen aktiviert wird (mit dem **Activation Interface**).

Diese Unterschiede sind oberhalb des RRL nicht sichtbar.



1.1.7. RMI Stubs und Skeletons

Der Stub/Skeleton Layer ist das Interface zwischen den Applikationen, dem Applikationslayer, und dem Rest des RMI Systems. Dieser Layer kümmert sich also nicht um den Transport; er liefert lediglich Daten an den RRL.

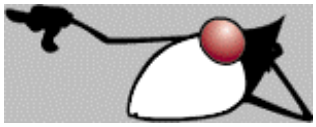
Ein Client, welcher eine Methode auf einem remote Server aufruft, benutzt in Wirklichkeit einen Stub oder ein Proxy Objekt für das remote Objekt, also quasi ein Ersatz für das remote Objekt.

Ein Skeleton für ein remote Objekt ist die serverseitige Grösse, welche die Methodenaufrufe an die remote Objektimplementation weiterleitet.

Die Kommunikation der **Stubs** mit dem clientseitigen RRL geschieht auf folgende Art und Weise:

- Der Stub (clientseitig) empfängt den Aufruf einer entfernten Methode und initialisiert einen Call, einen Verbindungsaufbau zum entfernten Objekt.
- Der RRL liefert eine spezielle Art I/O Stream, einen *'marshal'* (Eingabe/ Ausgabe) Stream, mit dessen Hilfe die Kommunikation mit der Serverseite des RRL stattfindet.
- Der Stub führt den Aufruf der entfernten Methode durch und übergibt alle Argumente an diesen Stream.
- Der RRL liefert die Rückgabewerte der Methode an den Stub.
- Der Stub bestätigt dem RRL, dass der Methodenaufruf vollständig und abgeschlossen ist.

Skeletons kommunizieren mit dem serverseitigen RRL auf folgende Art und Weise:



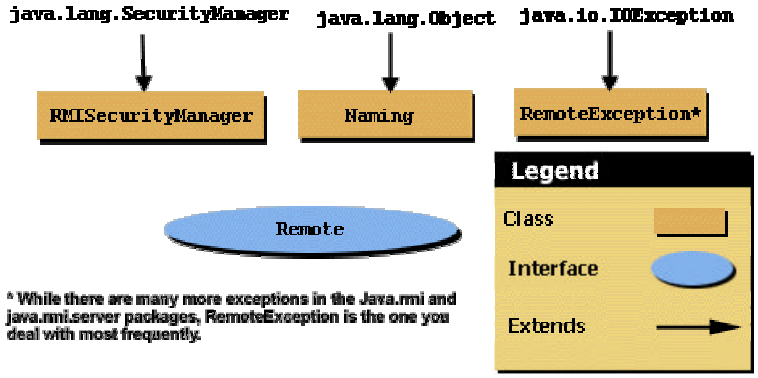
- Der Skeleton '*unmarshal*' (empfängt und interpretiert) alle Argumente aus dem I / O Stream, welcher durch den RRL aufgebaut wurde.
- Der Skeleton führt den Aufruf der aktuellen remote Objektimplementation durch.
- Der Skeleton '*marshals*' (interpretiert und sendet) die Rückgabewerte des Methodenaufrufes (oder einer Ausnahme, falls eine geworfen wurde) in den I / O Strom.

Bemerkung
Hüten Sie sich vor verteilten Deadlocks. Diese können beispielsweise auftreten, falls Sie versuchen Programmcodeböcke eines remote Objekts zu synchronisieren.
Falls Sie in einer verteilten Anwendung ein Lock, eine Sperre auf ein Objekt gesetzt haben, gibt es keinen Weg mehr zu unterscheiden, ob die Sperre von einem bestimmten Objekt oder irgendwelchen anderen Objekten stammt. Die Sperre kann somit im schlimmsten Fall auf ewig bestehen bleiben.

1.1.8. RMI Packages und Hierarchien

1.1.8.1. `java.rmi` Packages

Die grundlegenden Packages, Klassen und Interfaces, welche zur Entwicklung von RMI Clients und Servern eingesetzt werden, sind:



- `Naming`
  Diese Klasse ist `final` Klasse, mit deren Hilfe die RMI Clients und Server mit der Server Registry kommunizieren.
  Die Server Applikation benutzt die Methoden `bind` und `rebind`, um ihre Objektimplementationen bei der Registry zu registrieren (daher der Name dieses Lookup Servers 'Registry').
  Die Client Applikation benutzt die `lookup` Methode dieser Klasse, um eine Referenz auf das remote Objekt zu erhalten.

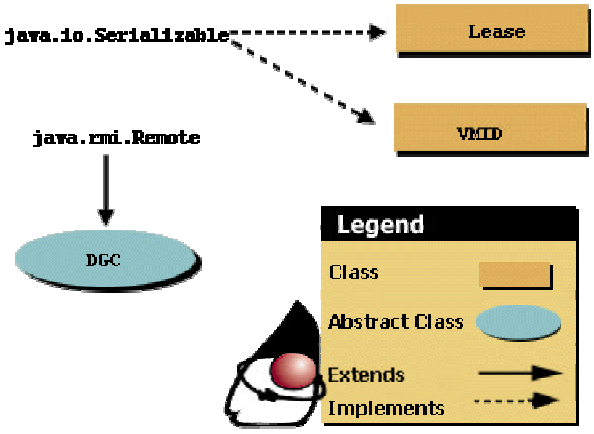
- `Remote` Interface
  Dieses Interface muss von allen Client Interfaces erweitert werden, welche auf das remote Objekt zugreifen möchten.

- `RemoteException`
  Diese Exception muss durch jede Methode geworfen werden, welche in einem remote Interface und Implementationsklassen definiert wird. Alle Clients müssen diese Exception abfangen.

- `RMISecurityManager`
  Diese kontrolliert den Zugriff auf lokale und remote Applikationen durch RMI Klassen und Interfaces.

1.1.8.2. Das `java.rmi.dgc` Package

Das `java.rmi.dgc` Package enthält Klassen, welche benötight werden, um remote Garbage Collection zu implementieren.

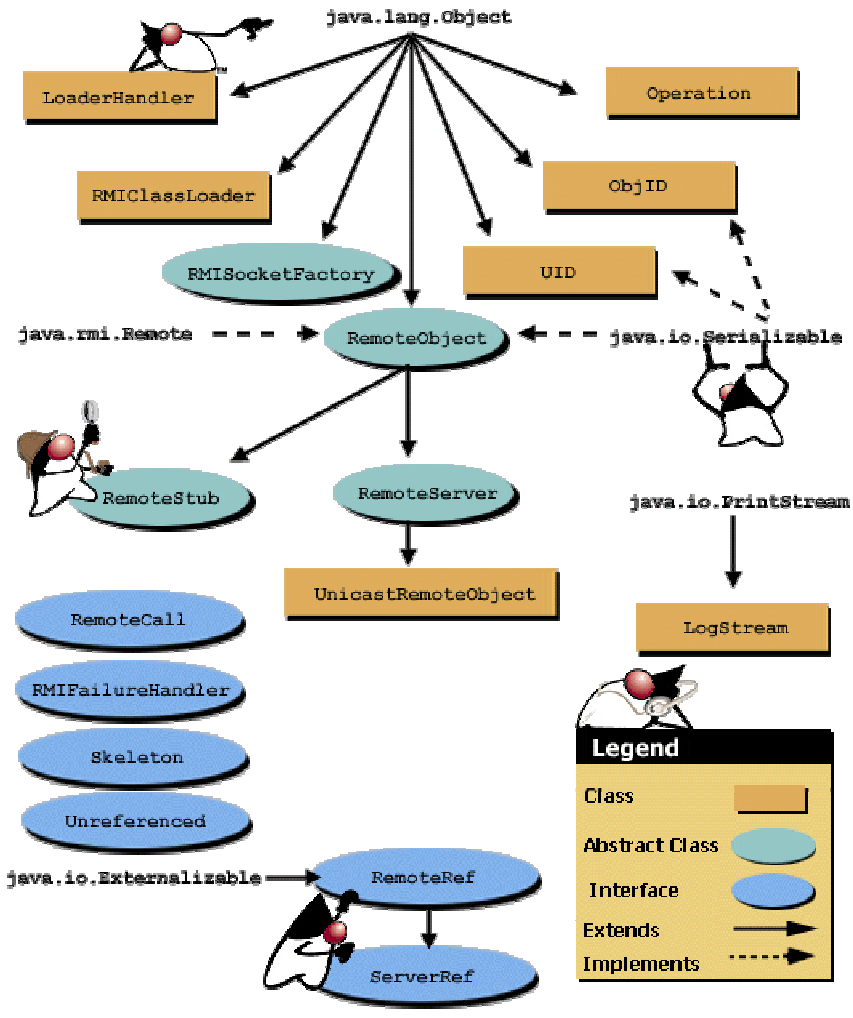


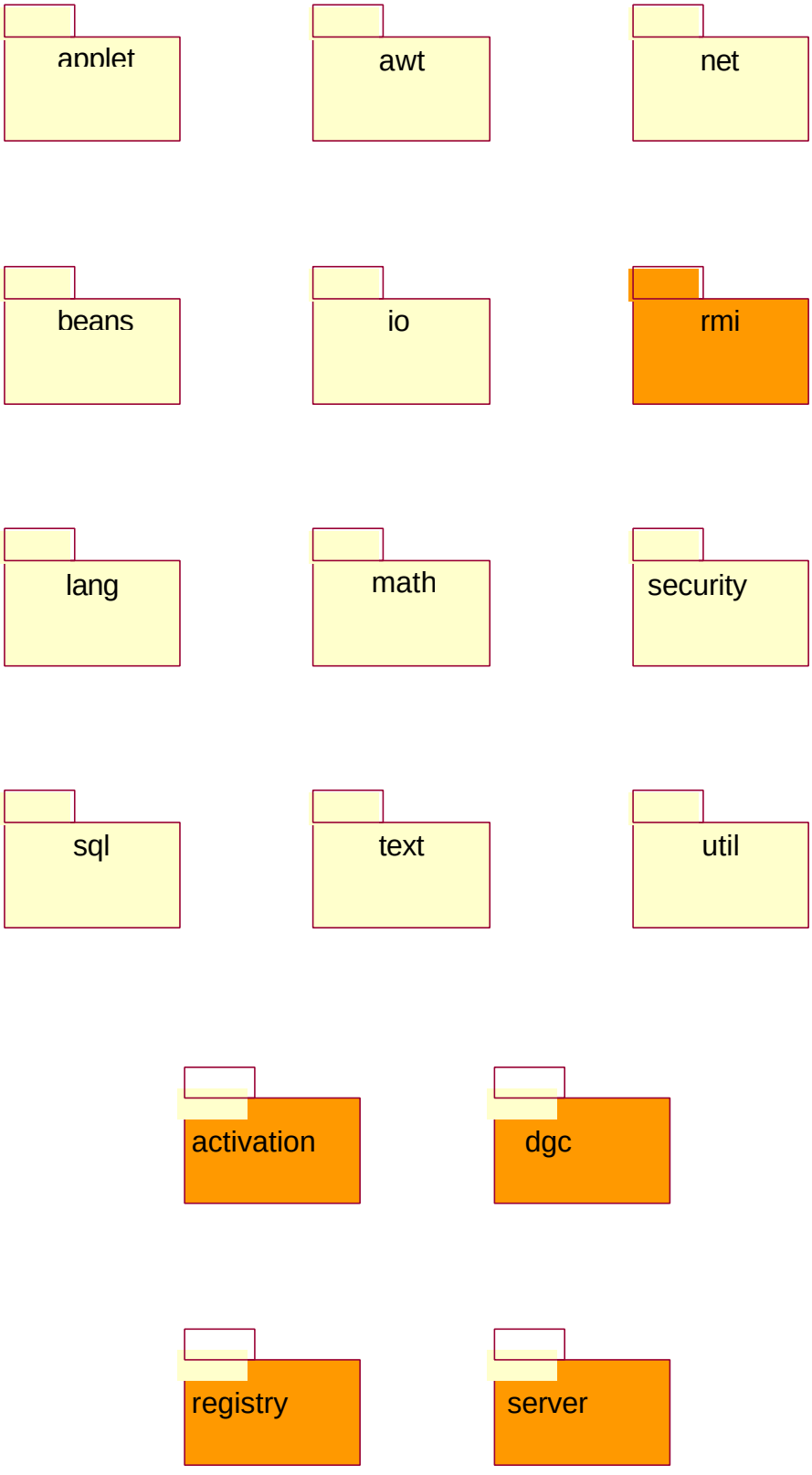
1.1.8.3. Das `java.rmi.server` Package

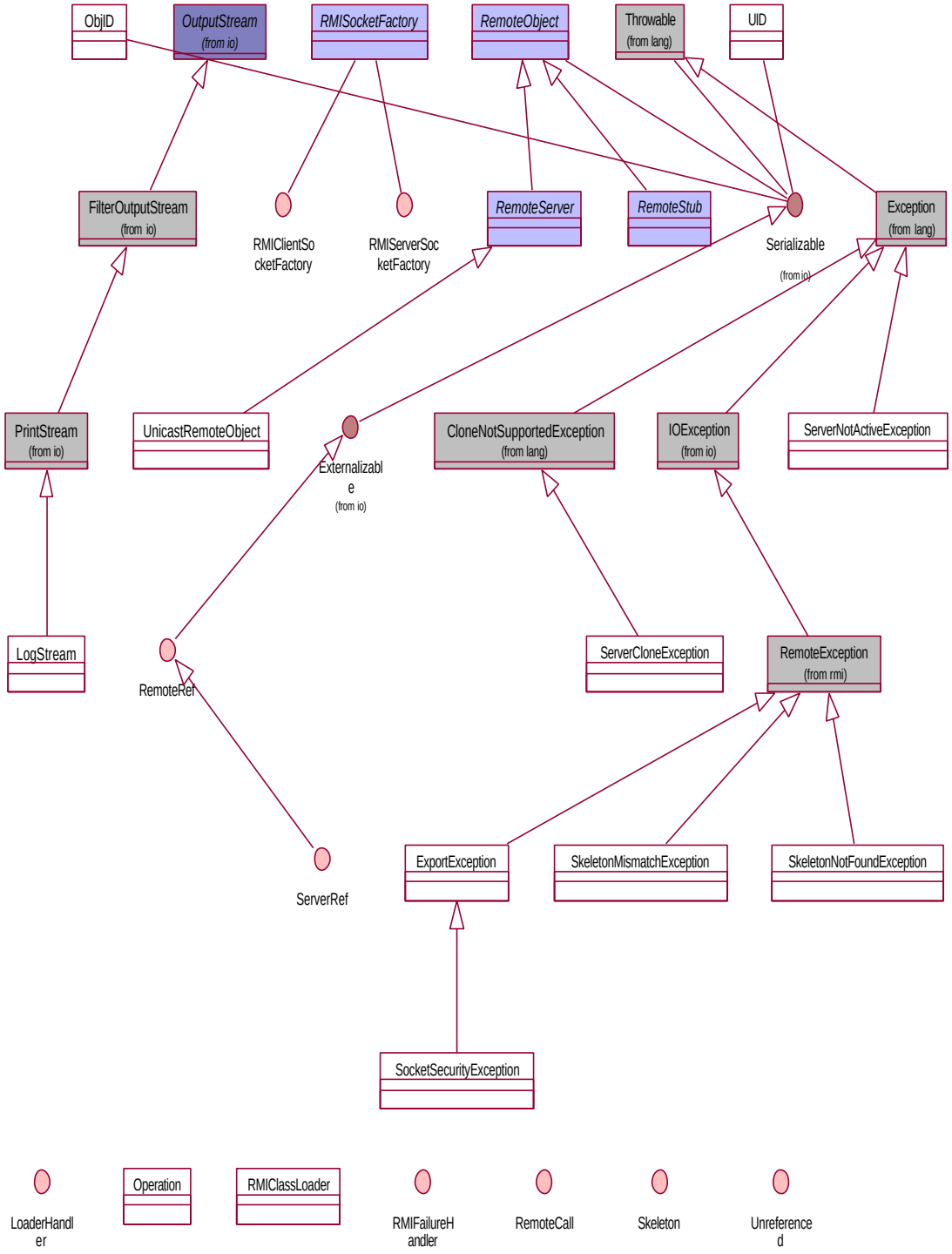
Aus der Vielzahl der Klassen betrachten wir zwei etwas genauer:

- `RMIClassLoader`
  Der RMI Class Loader ist der Class Loader, mit dem Stubs und Skeletons der remote Objekte (sowie Argumente und Rückgabewerte von remote Methoden) geladen werden. Falls der RMIClassLoader versucht Klassen vom Netzwerk zu laden wird eine Exception geworfen, falls kein Security Manager installiert wurde.

- `UnicastRemoteObject`
  Das Unicast Remote Objekt ist die Oberklasse für alle RMI Implementationsklassen.




applet
awt
net
beans
io
rmi
lang
math
security
sql
text
util
activation
dgc
registry
server


ObjID
OutputStream
(from io)
RMISocketFactory
RemoteObject
Throwable
(from lang)
UID
FilterOutputStream
(from io)
RMIClientSo
cketFactory
RMIServerSoc
ketFactory
RemoteServer
RemoteStub
Serializable
(from io)
Exception
(from lang)
PrintStream
(from io)
UnicastRemoteObject
Externalizabl
e
(from io)
CloneNotSupportedException
(from lang)
IOException
(from io)
ServerNotActiveException
LogStream
RemoteRef
ServerCloneException
RemoteException
(from rmi)
ServerRef
ExportException
SkeletonMismatchException
SkeletonNotFoundException
SocketSecurityException
LoaderHandl
er
Operation
RMIClassLoader
RMIFailureH
andler
RemoteCall
Skeleton
Unreference
d

1.1.9. Kreieren einer RMI Applikation

Das folgende Beispiel verwendet ein Bankkonto und einen (Bank-)Kontomanager, um zu illustrieren, wie RMI eingesetzt werden kann. Beteiligt ist also eine Bank, bei der Sie ein Konto eröffnen können. Die Konten werden von einem Kontomanager, einem Angestellten der Bank, verwaltet.

Nachdem Sie ein Konto eröffnet haben, können Sie Geld einzahlen, Geld abheben und den Kontostand abfragen.

Falls Sie dieses Problem mit RMI lösen möchten, könnten Sie zwei Interfaces definieren:

- Bankkonto.java

```
package rmi.bank;
interface Bankkonto extends Remote {
      public float abfragenKontostand();
      public void abheben(float betrag);
      public void einzahlen(float betrag);
}
```

- Kontomanager.java

```
package rmi.bank;
interface Kontomanager extends Remote {
      public Konto eröffnen(String name, float startKapital);
}
```

Hinweis
Diese Interfaces sind so definiert, dass Sie ein Konto nur mit Hilfe des Kontomanagers eröffnen können. Sie erhalten vom Kontomanager (Objekt) ein Konto (Objekt). Der Kontomanger liefert Ihnen eine aktuelle Instanz eines Kontos, falls dieses bereits existiert.

Diese Art, Objekte zu kreieren, ist ein Beispiel für den Einsatz eines bestimmten *'Design Pattern*' (Entwurfsmuster) in der objektorientierten Programmiermethodologie. Dieses Muster wird als **Factory Methode** bezeichnet.

Die Factory Methode gestattet es einem Objekt andere Objekte zu kreieren. Das ist in unserem Fall genau das, was wir benötigen.

Der Kontomanager kontrolliert das Anlegen eines neuen Kontos. Falls Sie in eine Bank gehen möchten und ein neues Konto eröffnen möchten, sollten Sie eine Hinweis darauf erhalten, ob Sie bereits (und allenfalls welche) ein Konto bei dieser Bank besitzen.

1.1.9.1. Ablauf zum Kreiern einer RMI Applikation

Um eine remote verfügbare Applikation in RMI zu kreieren, gehen Sie folgenermassen vor:

1. Definieren Sie die remote Objekte, mit denen gearbeitet werden soll, als Java Interfaces.
2. Kreieren Sie Implementationsklassen für die Interfaces.
3. Übersetzen Sie Interfaces und Implementationsklassen.
4. Kreieren Sie Stub und Skeleton Klassen mit Hilfe des `rmic` Befehls, angewandt auf die Implementationsklassen.
   Achtung:
   *bei CORBA oder einigen anderen Techniken müssen Sie **zuerst** die Interfacebeschreibung übersetzen und **dann** die Implementationsklassen kreieren (und den generierten Programmcode einbauen).*
5. Kreieren Sie eine Serverapplikation, welche die remote Objekte administriert, und übersetzen Sie die Serverapplikation
6. Starten Sie die `RMIregistry` (oder bauen Sie diese gleich selbst als Teil der Serverapplikation: in diesem Fall entfällt dieser Schritt).
7. Testen Sie den Client.

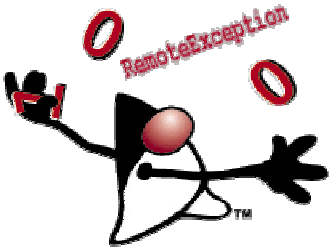
Das vollständige Beispiel für unser Problem sieht folgendermassen aus:

1.1.9.1.1. Das Konto Interface

```
package Bankkonto;

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

/**
 * Definition der Methoden für unser Bankkonto
 * abfragen des Kontostandes
 * einzahlen eines bestimmten Betrages
 * abheben eines bestimmten Betrages
 */

public interface Konto
    extends Remote
{
    /**
     * ebfragen des Kontostandes : liefert als float den Kontostand
     */
    public abstract float kontostand()
        throws RemoteException;

    /**
     * einzahlen eines (float) Betrages f auf das Konto
     */
    public abstract void einzahlen(float f)
        throws ungültigerBetragException, RemoteException;

    /**
     * abheben eines (float) Betrages f vom Konto
     */
    public abstract void abheben(float f)
        throws ungültigerBetragException, RemoteException;
}
```

Unser `Konto` Interface muss `java.rmi.Remote` erweitern und als `public` deklariert sein, um remote (für Clients in anderen virtuellen Maschinen) verfügbar zu sein.

Beachten Sie
Alle Methoden werfen die `java.rmi.RemoteException`. Diese Ausnahme wird immer dann geworfen, falls ein Problem beim Aufruf einer Methode eines remote Objekts auftritt.
Die Methoden `einzahlen(...)` und `abheben(...)` werfen zudem eine `ungültigerBetragException`, falls ein negativer Betrag einbezahlt oder mehr Geld abgehoben werden soll, als auf dem Konto ist.



Jede Methode in einem Remote Interface muss mindestens eine `RemoteException` (nicht eine Unterklasse von `RemoteException`) werfen. Dies wird von `rmic` überprüft.

1.1.9.1.2. Das KontoManager Interface

Das vollständige `KontoManager` Interface sieht folgendermassen aus:

```
package Bankkonto;

/**
 * Der Kontomanager kreiert (im Auftrag, als Objekt Factory) ein
 * neues Konto für einen Kunden
 * Parameter: Kundenname und Startbetrag.
 */

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

public interface KontoManager
    extends Remote
{

    /**
     * eröffnen eines Kontos für den Kunden s mit Anfangskontostand f
     */
    public abstract Konto eröffnen(String s, float f)
        throws ungültigerBetragException, RemoteException;
}
```

Argumente oder Rückgabewerte einer remote Methode können von irgend einem Java Datentyp sein, inklusive Objekte, solange diese Objekte entweder serialisierbar oder externalizable (im Falle einer `Externalizable` Klasse delegiert die Objekt Serialisierung die Kontrolle über das externe Format und die Art und Weise, wie die Supertypen gespeichert und wiederhergestellt werden, vollständig an das Objekt).

Auch dieses Interface erweitert `java.rmi.Remote` und die Methode `eröffnen(...)` kann die Ausnahme `RemoteException` (plus `ungültigerBetragException`, falls der Anfangsbetrag negativ ist) werfen.

Das `Konto` Interface wird von einer Klasse implementiert, welche alle deren Methoden implementiert und `UnicastRemoteObject` erweitert.

Hier hat sich eine Namenskonvention eingebürgert: alle Klassen, welche Interfaces implementieren, übernehmen den Namen des Interfaces plus dem Zustz "Impl".

```
// KontoImpl - Implementation des Konto Interfaces
//
// Instanzen dieser Klasse implementieren die Methoden:
// kontostand, abheben, einzahlen
// welche im Interface Konto definiert wurden
package Bankkonto;

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class KontoImpl extends UnicastRemoteObject implements Konto {
	// aktueller Kontostand
	private float kontostand = 0;
	// neues Konto eröffnen : Konstruktor
	public KontoImpl(float neuerKontostand) throws RemoteException {
		System.out.println("KontoImpl: Konstruktor");
		kontostand = neuerKontostand;
	}
	// Methoden
	// Kontostand abfragen
	public float kontostand() throws RemoteException {
		System.out.println("KontoImpl.kontostand(): Kontostand abfragen;
				Kontostand ="+kontostand);
		return kontostand;
	}
	// Betrag einbezahlen, sofern Betrag positiv
	public void einzahlen(float betrag) throws ungültigerBetragException, RemoteException {
		System.out.println("KontoImpl.einzahlen(): Betrag einzahlen;");
		System.out.println("KontoImpl: Kontostand alt: "+kontostand);
		if (betrag<0) {
			throw new ungültigerBetragException("Fehler :
					Versuch einen negativen Betrag einzubezahlen!");
		} else {
			kontostand = kontostand + betrag;
		}
		System.out.println("KontoImpl.einzahlen(): Kontostand neu: "+kontostand);
	}
	// Betrag abheben, falls das Konto darüber verfügt
	public void abheben(float betrag) throws ungültigerBetragException, RemoteException {
		System.out.println("KontoImpl.abheben():
			Betrag abheben; Betrag ="+betrag+" Kontostand alt: "+kontostand);
		System.out.println("KontoImpl.abheben(): Kontostand alt: "+kontostand);
		if (betrag < 0) {
			throw new ungültigerBetragException("KontoImpl.abheben() Fehler :
			     Versuch einen negativen Betrag abzuheben!");
		} else {
			if ((kontostand - betrag)<0) {
				throw new ungültigerBetragException("KontoImpl.abheben()
					Fehler : Versuch Konto zu überziehen!");
			} else {
				kontostand = kontostand - betrag;
			}
		}
		System.out.println("KontoImpl.abheben(): Kontostand neu: "+kontostand);
	}
}
```

Der Kontomanager wird durch eine Klasse implementiert, welche in der Lage ist, neue Konten anzulegen und die Konten abzuspeichern bzw. zu verwalten.

Konkret verwendet der Kontomanager einen Vektor als Speicher. Die Objekte werden mit Hilfe einer Klasse `KontoInfo` gebildet. Diese Klasse ist eine reine Hilfsklasse! Sie wird auch benutzt, um schnell nach bestehenden Konton suchen zu können.

```
package Bankkonto;

/**
 * Der Kontomanager kreiert (im Auftrag, als Objekt Factory) ein
 * neues Konto für einen Kunden
 * Parameter: Kundenname und Startbetrag.
 */
import java.io.PrintStream;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class KontoManagerImpl extends UnicastRemoteObject
    implements KontoManager
{

    private static Vector konten = new Vector();

    public KontoManagerImpl()
        throws RemoteException
    {
      System.out.println("KontoManager: Konstruktor");
    }
    public Konto eröffnen(String name, float startKapital)
        throws RemoteException, ungültigerBetragException
    {
      System.out.println("KontoManager.eröffnen():
                          Startkapital= "+startKapital);
            KontoInfo a;
            for(int i = 0; i < konten.size(); i++) {
                  a = (KontoInfo)konten.elementAt(i);
                  if(a.name.equals(name))
                  return a.konto;
            }
            if(startKapital < (float)0)
                  throw new ungültigerBetragException("Fehler :
                              negativer Startbetrag!");
            a = new KontoInfo();
            try {
                  a.konto = new KontoImpl(startKapital);
            }
            catch(RemoteException e) {
                  System.err.println("Fehler :
                              neues Konto konnte nicht kreiert werden!");
                  System.err.println(" " + e.getMessage());
                  throw e;
            }
            a.name = name;
            konten.addElement(a);
            return a.konto;
    }
}
```

Die `KontoInfo` Klasse ist eine Art Container Klasse, welche vom KontoManager (der Implementationsklasse davon) eingesetzt wird.

```
package Bankkonto;
// Container Klasse zum Verwalten der Konten
class KontoInfo {
      String name;
      KontoImpl konto = null;
}
```

Der Klassenpfad ist eine der Knackpunkte für verteilte Java Applikationen!
Sie sollten sich überlegen, die -d Option beim übersetzen der Java Programme einzusetzen, um die Klassendateien klar zu lokalisieren.

In unseren Beispielen werden deswegen in der Regel die BAT Dateien explizit angegeben und zusätzlich der CLASSPATH auf vernichtet (auf leer gesetzt):

```
javac -d . *.java
```

Mit diesem Flag werden auch die Package Verzeichnisse angelegt. In den BAT Dateien wird dies anders gelöst.

Nachdem Sie die Interfaces implementiert haben, müssen Sie Stubs und Skeletons kreieren. Mit diesen wird auf die jeweiligen remore Objekte zugegriffen. Dies geschieht mit Hilfe des `rmic`, des RMI Compilers. Dies geschieht nachdem die Implementationen geschrieben wurden, aber bevor die Server Applikation gestartet wird.

```
rmic  Optionen Package.InterfaceImpl ...
```

Zum Beispiel:

```
@echo off
cd ..
echo RMI Compile Bankkonto.KontoImpl
rmic -classpath . -keep Bankkonto.KontoImpl
echo RMI Compile Bankkonto.KontoManagerImpl
rmic -classpath . -keep Bankkonto.KontoManagerImpl
cd bankkonto
echo Kreieren eines Archives (Bankkonto.jar)
jar cvf Bankkonto.jar *_*.class
copy Bankkonto.jar ..\*.*
echo Das Archives (Bankkonto.jar) steht in den Verzeichnissen Banken und
Bankkonto
pause
```

Der RMI Compiler kreiert vier zusätzliche Dateien:

```
KontoImpl_Skel.java
KontoImpl_Stub.java
KontoManagerImpl_Skel.java
KontoManagerImpl_Stub.java
```

Bemerkung
Falls Sie eine Implementationsklasse ändern, müssen Sie diese neu übersetzen, der RMI Compiler muss Stubs und Skeletons neu generieren, die Registry muss gestoppt und neu gestartet werden und der Server ebenfalls.

Der Bankserver (`BankServer.java`) ist eine Applikation, welche die Kontomanager Implementationsklasse (`KontoManagerImpl.java`) jeweils bei Kundenanfragen instanziert und dem Client zur Verfügung stellt.

```
1. // Bank Server - Klasse, welche den RMI Server darstellt
2. //
3. package Bank;
4. import java.rmi.*;
5. import Bankkonto.*;

6. public class BankServer {
7. public static void main(String args[] ) {
8. System.out.println("BankServer :
                          start und setzen des SecurityManagers");
9. // kreieren und installieren eines SecurityManagers
10.   //System.setSecurityManager(new RMISecurityManager() );
11.   try {
12.   // Instanzen : Objekte,welche registriert werden
13.   System.out.println("BankServer.main() :
                                    kreieren eines KontoImpl Objekts");
14.   KontoManagerImpl kmi = new KontoManagerImpl();

15.   // binden der Instanz an die Registry
16.   System.out.println("BankServer.main() :
              binden des KontoManagers an den Bankserver [Registry]r");
17.   // zum Testen : Abfrage der Registry
18.   for (int i=0; i< Naming.list("rmi://localhost/").length; i++)
19.   System.out.println(" "+Naming.list("rmi://localhost/")[i]);

20.   Naming.bind("KontoManager",kmi);
21.   System.out.println("KontoManager Server ist gebunden!");
22.   } catch (Exception e) {
23.   System.err.println("BankServer.main() :
                    eine Ausnahme wurde geworfen - "+e.getMessage() );
24.   e.printStackTrace();
25.   }
26.   System.out.println("BankServer:main()... und Tschuess!");
27.   }
28.   }
```

Der Server publiziert' die Instanz der Kontomanager Implementationsklasse, indem er das Objekt mit einem Namen verbindet, der in einer Loopup Applikation abgespeichert wird, der rmiregistry.
Sie könnten diese Registry auch direkt im Anwendungsprogramm kreieren:

```
LocateRegistry.createRegistry( PORT );
```

Das "Binden" geschieht auf Zeile 20 oben:
`Naming.bind("KontoManager",kmi);`

(`java.rmi.Naming.rebind(...)`).

Diese Methode ordnet dem Namen "KontoManager" das Objekt `kmi` zu, sie "bindet" die beiden zusammen. Dabei wird jede bereits vorhandene Bindung mit dem selben Namen in der Registry einfach überschrieben.

Die `Naming` Klasse verfügt über zwei Methoden, mit denen Objekte an Namen gebunden werden können: `bind` und `rebind`. Der Unterschied ist der, dass im Falle von `bind` die Exception `java.rmi.AlreadyBoundException` geworfen wird, falls das Objekt bereits registriert ist.

Argument für bind und rebind sind URL ähnliche Zeichenketten und der Name der Instanz der Objektimplementation (siehe oben für ein Beispiel). Das Format des URL Strings ist :

```
rmi://<i>host:port/name
```

wobei:
`rmi` das Protokoll,
`host` der Name des RMI Servers (DNS Notation),
`port` die Portnummer ist, auf der der Server auf Anfragen wartet,
`name` der exakte Namen ist, den der Client im Aufruf `Naming.lookup` verwenden muss, falls er dieses Objekt benötigt.

Falls das Protokoll nicht angegeben wird, ist der Standardwert `rmi,` bei `host` ist der Standardwert `localhost` und bei Port `1099`.

Aus **Sicherheitsgründen** kann eine Applikation lediglich an eine Registry gebunden werden, welche lokal auf dem selben Rechner läuft.

Die RMIRegistry ist eine Applikation, welche einen einfachen Namens- Lookup Dienst zur Verfügung stellt. In unserem Beispiel stellt die Applikation der RMI Registry einen Namen und eine Objektreferenz zur Verfügung. Wie Sie oben gesehen haben, besteht das Programm aus wenigen Zeilen. Falls Sie die Registry in den Server integrieren, besteht die RMI Registry aus genau einer Zeile. Die RMI Registry liefert dem Garbage Collector auch Informationen über Referenzen. Das Programm `rmiregistry` muss laufen, bevor der Server gestartet wird und versucht seine remote Objekte zu binden:

```
rmiregistry
java -classpath . -Djava.security.policy=java.policy Bank.BankServer
```

Bemerkung
Auf der Win32 Plattform können Sie die Registry auch mit `start rmiregistry` starten. Schauen Sie sich in einem DOS Fenster die Syntax an. Damit haben Sie einige Optionen in Bezug auf Prioritäten, schliessen des DOS Fensters usw.

Auf der Kommandozeile (siehe oben) können Sie auch Properties setzen. Im obigen Beispiel wurde die Security Policy Datei auf diese Art und Weise angegeben. Weitere Optionen wären:

- Die Angabe der Codebase:
  `java.rmi.server.codebase`
  gibt die URL an, von der die benötigten Klassen heruntergeladen werden können, falls sie benötigt werden.
- Falls Sie eine Logdatei mitführen möchten, könenn Sie den Property Parameter `java.rmi.server.logCalls` auf `true` setzen. Standardmässig ist dieser Wert auf `false` gesetzt. Die Ausgabe erfolgt auf stderr.

Zwei Beispiele für das Einschalten des Loggings wären:

```
java -Djava.rmi.server.logCalls=true bank.BankServer

java -classpath . -Djava.security.policy=java.policy
                  -Djava.rmi.server.logCalls=true
                                                  Bank.BankServer
```

Nachdem Sie die Implementation angemeldet haben 'exportiert' die Registry dieses Objekt: sie bietet es Clients an. Der Client kann eine URL an die Registry senden. Die Registry liefert dann eine Referenz auf das remote Objekt. Der Lookup geschieht mittels eines Aufrufes:

```
Naming.lookup(...);
```

wobei als Argument eine Zeichenkette mitgegeben wird, den URL.

```
rmi:// host:port/name
```

Als nächstes schauen wir uns nun an, wie der Client diese Information verwertet!
Der Bankkunde (`Kunde.java`) versucht ein `Kontomanager` Objekt mittels eines Registry Lookups zu finden. Die Registry befindet sich auf dem Host und am Port gemäss URL Angabe, welche an `Naming.lookup(...)` übergeben wurde. Das Objekt, welches zurückgegeben wird, muss noch in ein `Kontomanager` Objekt gecastet werden. Es kann dann eingesetzt werden, um ein Konto auf einen Namen mit einem bestimmten Startbetrag zu eröffnen und Bankgeschäfte zu starten (Einzahlen, Abhabe, ...).

```
// BankClient - Test Programm für unser RMI Bankkonto Beispiel
//
// Diese Klasse sucht ein "KontoManager" RMI Objekt,
// bindet dieses und öffnet eine KontoManager Instanz
// auf dem Server
//
// Dann legt es ein Konto mit demNamen <name> und einem oder
// keinem Startkapital an
//
// Das Konto wird dann getestet, indem einfach Beträge einbezahlt
// und abgehoben werden, sowie der Kontostand abgefragt wird.
//
package Bankkunde;

import java.rmi.*;
// Interfaces importieren
import Bankkonto.*;
import Bankkonto.*;
```

```
public class Kunde {
      public static void main(String args[] ) {
            // Argumente prüfen
            if (args.length < 2) {
                  System.err.println("Usage : ");
                  System.err.println("java Bankkunde
                              <server> <Kontonamen> [Startkapital]");
                  System.exit(1);
            }

            // kreieren und installieren des SecurityManagers
            System.setSecurityManager(new RMISecurityManager() );

            // Kontomanager suchen
            try {
                  System.out.println("Kunde.main() ; lookup KontoManager");
                  String url = new
                                    String("rmi://"+args[0]+"/KontoManager");
                  KontoManager ktm = (KontoManager)Naming.lookup(url);
                  // Initialisieren des Bankkontos
                  float startBetrag = 0.0f;
                  // ... falls ein Betrag angegeben wurde
                  if (args.length ==3) {
                        Float F = Float.valueOf(args[2]);
                        startBetrag = F.floatValue();
                  }

                  // ... und nun das Konto anlegen oder nachsehen
                  Konto konto = ktm.eröffnen(args[1], startBetrag);

                  // Jetzt spielen wir mit dem Konto
                  System.out.println("Kunde.main() :
                        aktueller Kontostand = "+konto.kontostand() );

                  System.out.println("Kunde.main() : 50.00 abheben");
                  konto.abheben(50.00f);

                  System.out.println("Kunde.main() :
                        aktueller Kontostand = "+konto.kontostand() );

                  System.out.println("Kunde.main() : 100.00 einzahlen");
                  konto.einzahlen(100.00f);

                  System.out.println("Kunde.main() :
                        aktueller Kontostand = "+konto.kontostand() );

                  System.out.println("Kunde.main() : 25.00 einzahlen");
                  konto.einzahlen(25.00f);

                  System.out.println("Kunde.main() :
                        aktueller Kontostand = "+konto.kontostand() );
            } catch (Exception e) {
                  System.err.println("Kunde :
                        eine Ausnahme wurde geworfen; "+e.getMessage());
                  e.printStackTrace();
            }
            System.exit(1);
      }
}
```

Nachdem das Konto angelegt wurde, lassen wir den Client noch einige Operationen ausführen. Natürlich sollten diese Operationen mit einem GUI erledigt werden können. Aber dafür hatte ich bisher noch keine Zeit.

Den Bankclient können Sie auf irgend einem Rechner starten, welcher mit dem Server über TCP/IP verbunden ist. Zusätzlich muss derZugriff auf die Klassendateien des Clients, dessen Stub und Interfaces garantiert sein (sonst können Sie den Client schlecht starten).

```
@echo off
Rem
Rem modifiziert
Rem
java -cp . -Djava.security.policy=java.policy Bankkunde.Kunde localhost
Joller 10000
Rem : jetzt folgt die Ausgabe
pause
```

Falls Sie die Registry und den Bankserver gestartet haben, liefert dieser Aufruf Ihnen folgende Ausgabe (vermutlich möchten Sie ein eigenes Konto):

```
Kunde.main() ; lookup KontoManager
Kunde.main() : aktueller Kontostand = 10000.0
Kunde.main() : 50.00 abheben
Kunde.main() : aktueller Kontostand = 9950.0
Kunde.main() : 100.00 einzahlen
Kunde.main() : aktueller Kontostand = 10050.0
Kunde.main() : 25.00 einzahlen
Kunde.main() : aktueller Kontostand = 10075.0
Taste drücken, um fortzusetzen . . .
```

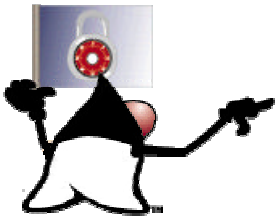
Beim zweiten Aufruf des selben Clients sieht die Situation anders aus: jetzt sollte das Konto bereits angegelegt sein und auch ein Kontostand vorliegen, der sich vom Anfangskontostand (1000.00) unterscheidet. Der zweite Aufruf liefert folgende Ausgabe:

```
Kunde.main() ; lookup KontoManager
Kunde.main() : aktueller Kontostand = 10075.0
Kunde.main() : 50.00 abheben
Kunde.main() : aktueller Kontostand = 10025.0
Kunde.main() : 100.00 einzahlen
Kunde.main() : aktueller Kontostand = 10125.0
Kunde.main() : 25.00 einzahlen
Kunde.main() : aktueller Kontostand = 10150.0
Taste drücken, um fortzusetzen . . .
```

Das Witzige am Kundenprogramm ist, dass Sie dieses Verhalten eigentlich im Programmcode nicht sehen. RMI erledigt diese Arbeit für Sie!

1.1.10. RMI Security

Um den RMI ClassLoader benutzen zu können, muss ein Security Manager installiert sein.



Dieser überprüft alle Klassen, welche über das Netzwerk geladen werden in Bezug auf Sicherheit und Sicherheitsverletzungen. Falls Sie keinen Security Manager starten / laden, kann Ihre Applikation keine Klassen über eine Netzwerkverbindung von einem anderen Host laden.

Beim Starten einer RMI Applikation können Sie zwei sicherheitsrelevante Parameter, Properties, setzen.

1) `java.rmi.server.codebase`, eine URL, welche angibt, von wo der Client die Klassen herunterladen muss. Natürlich muss der Server diese Adresse auch kennen und darauf Zugriff haben. Wir werden später im Praxisteil ein Beispiel kennen lernen, bei dem explizit mit Hilfe eines sehr einfachen HTTP Servers, bestehend aus im Wseentlichen zwei Java Klassen, die Dateien zum Client und vom Client auch zum Server verschiben werden.

2) Falls `java.rmi.server.useCodebaseOnly` auf `true` gesetzt wird, können Klassen nicht mehr über URLs, die der Client angibt, geladen werden (das Laden der Klassen ist dann nur noch über die Codebase möglich).

Falls der RMI Client ein Applet ist, verlangt der Browser bereits über seinen Security Manager und Class Loader die Sicherheit, die aus Sichtdes Browsers, zusammen mit den Policies auf Benutzerebene nötig sind.

Falls der Client eine Applikation ist, werden lediglich die remote Interface Definitionen, Stub Klassen und die Erweiterungen davon, herunter geladen. Falls ein Client zusätzliche Klassen vom Server laden möchte, dann kann er die Methode

```
RMIClassLoader.loadClass(...)
```

zusammen mit der selben URL, die bereits beim Aufruf von `Naming.lookup` verwendet wurde.

Der Transport Layer versucht normalerweise eine direkte Socket Verbindung vom Client zum Server aufzubauen. Falls dies nicht möglich ist, versucht die `createSocket` Methode der `java.rmi.server.RMISocketFactory` Klasse, eine Hypertext Transfer Protocol (HTTP) Verbindung aufzubauen, um RMI Calls als einen HTTP POST Request zu senden.

Falls die `createServerSocket` Methode eine neue Verbindung als eine HTTP Verbindung erkennt, wird die Antwort auf den POST Request wieder passend umgewandelt.

Auch für die Kommunikation durch eine Firewall werden keine speziellen Konfigurationen benötigt.

Bemerkung
Die Client Applikation kann verbieten, dass RMI Calls als HTTP Requests weitergeleitet werden. Dies geschieht durch das Setzen der

```
java.rmi.server.disableHTTP
```

Property als `true`.

1.1.11. Übung - Bauen einer Remote Method Invocation Applikation

In dieser einfachen Übung schauen Sie lediglich ein Programm an, welches eine Flugreservation durchführt. Die Applikation wurde mit RMI implementiert.

Der Kunde kann dabei die Angaben in ein GUI eingeben. Die Flugdaten werden dann zu einem remote Server gesandt, auf eine remote JVM.

Die Daten werden in einer Datenbank abgespeichert. Zudem stehen verschiedene Methoden zur Verfügung, mit deren Hilfe die Daten abgefragt, manipuliert und gelöscht werden können.

Der Kunde gibt zum Beispiel Daten zum Flug (Abflugshafen, Zielflughafen, Flug) ein. Alle persistenten Daten werden durch die Klasse `Datenbank.java` verwaltet.

Und hier eine mögliche Liste der Methoden für die Datenbank:

```
public KundenInfo [] liesKunde(String nName, String vName)
public void neuerKunde (String id, String nName, String vName, String addr)
public void bestehenderKunde(String id, String nName, String vName, String addr)
public void taetigeReservation(String id, String flug, String confirm, String sKlasse)
public String [] bestimmeAbflughafen()
public String [] bestimmeDestination() {
public boolean   legeSitzFest (KundenReservationsInfo res)
public FlugInfo [] bestimmeFlug(String origin, String dest, String datum)
```

Als Nächstes schauen wir uns die drei wichtigsten Interfaces an, die wir implementieren müssen:

- Kunden Interface
- KundenReservation Interface
- Flug Interface

Das Kunden Interface ist eines der wichtigsten Interfaces dieses Programms. Es erweitert `Remote` und alle seine Methoden können `RemoteException` werfen, was Bedingung für alle remote Server sind.

```
package flugreservation;

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

/**
 * Title:
 * Description:

 * @author J.M.Joller
 * @version 1.0
 */

public class Kunde_Impl extends UnicastRemoteObject implements Kunde {

      private Datenbank db = null;

      // Konstruktor
      public Kunde_Impl(Datenbank db) throws RemoteException {
            this.db = db;
      }

      // Kundenrecord
      public KundenInfo liesKundenInfo(String kundenID) throws
                                    RemoteException {
            KundenInfo kunde = null;
            try {
                  kunde = db.liesKunde(kundenID);
            } catch(RecordNotFoundException e) {
                  System.err.println("liesKunde Exception : "+e);
            }
            return kunde;
      }

      // Liste der möglichen Kunden (Vorname, Name)
      public KundenInfo[ ] liesKundenInfo(String vorName, String nachName)
                                          throws RemoteException {
            KundenInfo[ ] kunde = null;
            try {
                  kunde = db.liesKunde(vorName, nachName);
            } catch(Exception e) {
            System.err.println("liesKundenInfo(array) Exception: "+e);
            }
            return kunde;
      }
      // Eintrag eines Kundenrecords in die Kundentabelle
      public void neueKundenInfo(String id, String nachName,
         String vorName, String strassenAdresse) throws RemoteException {
            try {
                  db.neuerKunde(id, nachName, vorName, strassenAdresse);
            } catch(InvalidTransactionException e) {
                  System.err.println("neuerKunde Exception : "+e);
            } catch(DuplicateIDException e) {
                  System.err.println("neuerKunde Exception : "+e);
            }
      }
```

```
      public void bestehendeKundenInfo(String id, String nachName,
         String vorName, String strassenAdresse) throws RemoteException {
            try {
                  db.bestehenderKunde(id, nachName, vorName,
                                                      strassenAdresse);
            } catch(RecordNotFoundException e) {
                  System.err.println("bestehendeKundenInfo Exception :
                        "+e);
            }
      }

      // produziere eine neue Customer ID, mittels einer Formel,
      // die nur ein Programmierer kennen kann
      public String produziereID(String nachName, String vorName) throws
                                                      RemoteException {
            char EN = Character.toUpperCase(vorName.charAt(0) );
            char LN = Character.toUpperCase(nachName.charAt(0) );
            int ID = (int)(Math.random()*100000);
            String kundenID = new String(
                  new
         StringBuffer().append(EN).append(LN).append(Integer.toString(ID)
));
            return kundenID;
      }

      // KundenInfo Objekt als String
      public String produziereKundenString(KundenInfo info) throws
                                    RemoteException {
            String KundenString = info.kundenID + "-" + info.nachName + "-"
                              + info.vorName + "-" + info.strassenAdresse;
            return KundenString;
      }
}
```

Das dritte Interface beschreibt die Flüge. Auch dieses Interface erweitert `Remote` und wirft `RemoteExceptions`.

```
package flugreservation;

import java.rmi.*;
import java.rmi.server.*;
/**
 * Title:
 * Description:

 * @author J.M.Joller
 * @version 1.0
 */

public class Flug_Impl extends UnicastRemoteObject implements Flug {

    private Datenbank db = null;

    // Konstruktion mit super(),
    public Flug_Impl (Datenbank db) throws RemoteException {
        this.db = db;
    }

    // Liste der Abflughäfen
    public String [] bestimmeAlleAbflughafen () {
      String [] originStaedte = db.bestimmeAbflughafen();
      return originStaedte;
    }

    // Liste der Destinationen
    public String [] bestimmeAlleDestinationen () {
      String [] destStaedte = db.bestimmeDestination();
      return destStaedte;
    }

    // FlugInfo Objekte
    public FlugInfo[] bestimmeVerfuegbareFluege (String origin,
                String dest, String datum) {
      try {
          FlugInfo[] verfuegbareFluege = db.bestimmeFluege(origin,
                dest, datum);
          return verfuegbareFluege;
      } catch (Exception e) {
          System.err.println("Fehler beim Bestimmen der verfügbaren
                              Flüge in KundenReservation_Impl " + e);
      }
      return null;
    }

    // Zeichenkette mit der Fluginfo
    public String produziereFlugString (FlugInfo flug) {
        String FlugString = flug.flugID + "-" + flug.originStadt +
          "-" + flug.destStadt + "-" + flug.abflugDatum + "-" +
            flug.abflugZeit + "-xxx-xxx-xxx-" +
          flug.flugzeugID;

      return FlugString;
    }
}
```

Auch das Kundenreservations Interface wird remote implementiert.

```
package flugreservation;

import java.rmi.*;
import java.rmi.server.*;
/**
 * Title:
 * Description:

 * @author J.M.Joller
 * @version 1.0
 */

public class KundenReservation_Impl extends UnicastRemoteObject implements KundenReservation {

   private Datenbank db = null;

   // Aufruf von super(),
   // super exportiert das Objekt
   KundenReservation_Impl (Datenbank db) throws RemoteException {
         this.db = db;
   }

   // neue Reservation einfügen
   public void festlegenEinerReservation (String kundenID, String flugNummer, String
         bestaetigungsNummer, String serviceKlasse) throws RemoteException {

         try {
            db.taetigeReservation(kundenID, flugNummer, bestaetigungsNummer, serviceKlasse);
      } catch (InvalidTransactionException e) {
         System.err.println ("Exception in festlegenEinerReservation : " + e);
         }
   }

}
```

Wie geht's weiter?
Falls alle Interfaces definiert und implementiert sind:

1. übersetzen aller Klassen zum Testen und Debuggen der RMI Implementationen.
2. Kreieren von Stubs und Skeletons, welche zur Implementation der RMI Applikation benötigt werden.
3. Starten der RMI Registry.
4. Starten des JVM Servers.
5. Einsatz der Applikation mit einem GUI.

Dabei geht es um folgende Kommandos:

```
javac -d . *.java
```

Zusätzlich müssen wir die Stubs und Skeletons kreieren:

```
rmic -d . flugreservation.Kunde_Impl
rmic -d . flugreservation.Flug_Impl
rmic -d . flugreservation.KundenReservation_Impl
```

Falls Sie nun die Applikation testen möchten, müssen Sie als erstes die RMI Registry starten:

```
rmiregistry
```

dann den Server:

```
java -D java.rmi.server.codebase=/...
```

und schliesslich die Applikation:

```
appletviewer Airline.html
```

1.1.11.1. Aufgabe

Die folgende Aufgabe muss abgegeben werden.

Vervollständigen Sie die Flugreservationsskizze zu einer lauffähigen Applikation.
a) unter Zuhilfenahme von Sockets
b) als RMI Applikation

Beide Male sollten Sie ein GUI dazu entwerfen und natürlich möglichst viel Programmcode wiederverwenden.

Hinweis: diese Aufgabe entspricht in etwa einer Teilprüfung der (ganztägigen) Java Developer Certification.

1.1.12. Fragen - Quiz

Nach dem Durcharbeiten der Unterlagen sollten Sie in der Lage sein, die folgenden Fragen zu beantworten:

1. Welche der folgenden Aussagen über RMI ist korrekt?
   RMI gestattet es dem Entwickler, Programme zu schreiben, welche auf entfernte Objekte zugreifen und das genau so, als ob die Objekte lokal wären.

   Antwort:
   Diese Aussage ist teilweise korrekt!

2. RMI abstrahiert die Socket Verbindung und Datenströme für die Kommunikationzwischen Hosts.

   Antwort:
   Diese Aussage ist teilweise korrekt!

3. Beide der obigen Antworten sind korrekt!

   Antwort:
   Perfekt.

4. Als Entwickler einer RMI Applikation sind Sie für die Entwicklung einer Java RMI Interface Definition und deren Implementierungen zuständig, inklusive Stubs und Skeleton Generierung.

   a) trifft zu.
   b) Sie müssen sich auch noch um den Transport und die Verfolgung der remote Referenzen kümmern.

   Antwort:
   a) korrekt.
   b) Der RRL und der Transport Layer sind im RMI System bereits vorhanden und kümmern sich um den Transport und die Verfolgung der remote Objekte.

5. Welche der folgenden Aussagen über den Remote Reference Layer (RRL) trifft **nicht** zu?

   a) Der RRL managed die Kommunikation zwischen Stubs/ Skeletons und den unteren Transport Layern mit Hilfe eines spezifischen Remote Reference Protokolls, welches von Stubs und Skeletons unabhängig ist.

   b) Die clientseitige Komponente enthält Informationen über den Server und den Client und kommuniziert mit dem Transport Layer zur Serverseite.

   c) Der RRL definiert Endpunkte, mit deren Hilfe eine Verbindung in die Adressräume aufgebaut werden, in denen die remote Objekte vorhanden sind

   Antwort: c) ist die falsche Aussage : dies ist Aufgabe des Transport Layers.

   d) Der RRL ist verantwortlich für das Management der remote Referenzen, also der Referenzen auf remote Objekte. Dazu gehören auch Strategien, mit deren Hilfe Verbindungsunterbrüche überbrückt werden können, falls das Objekt plötzlich nicht mehr erreichnar sein sollte.

6. Welche der folgende Aussagen beschreibt nicht, wie Skeletons mit dem serverseitigen RRL kommunizieren?

   a) Das Stub/Skeleton führt einen remote Methodenaufruf aus und übergibt alle Argumente des Aufrufes an den Stream.

   Antwort : korrekt
   Der Stub führt den remote Methodenaufruf aus und übergibt alle Argumente an den Stream. Er arbeitet mit dem Skeleton zusammen..

   b) Das Skeleton unmarshals alle Argumente aus dem Stream I/O , welcher durch den RRL aufgebaut wurde.

   c) Das Skeleton führt den Up-Call zur aktuellen Server Implementation durch.

7. Wer ist dafür verantwortlich, dass die über das Netzwerk geladenen Klassen die Java Security Anforderungen erfüllen?

   a) Class Loader
   Antwort:
   Denken Sie nochmals nach! Der Class Loader lädt die Klassen.

   b) SecurityManager
   Antwort:
   Korrekt. Damit der RMI Class Loader überhauüt aktiv werden kann, muss der ein Security Manager installiert sein.

   c) RMI Class Loader
   Antwort:
   Siehe oben.

1.1.13. Zusammenfassung - Remote Methoden Invocation

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- die RMI Architektur beschreiben zu können, inklusive ihren Layern und dem Garbage Collection Prozess.
- einen RMI Server und Client in Java zu implementieren.
- Stubs und Skeletons mit Hilfe von RMI Compilern zu generieren
- zu definieren, welche Aufgabe die RMI Registry hat und wie sie grundsätzlich arbeitet.
- einige der Sicherheitsfragen im Zusammenhang mit RMI, dem Laden von Klassen über ein Netzwerk, zu beschreiben.
- eine RMI Aufgabe zu lösen.