In diesem Kapitel:

Java Remote Methode Invocation

1.1. Einführung ins Thema

Verteilte Systeme bedingen, dass Berechnungen, die in unterschiedlichen Adressräumen (Speicherbereichen) ablaufen, unter Umständen auf unterschiedlichen Maschinen, miteinander kommunizieren können. Java stellt als Standardwerkzeug für die Kommunikation die Sockets zur Verfügung. Sockets sind flexibel und für die allgemeine Kommunikation ausreichend. ABER : beim Einsatz von Sockets muss man Client- und Server- seitig Applikationsprotokolle schreiben, um Meldungen, die ausgetauscht werden sollen, zu kodieren und zu dekodieren. Die Definition eigener Protokolle ist, wie wir gesehen haben, nicht so ganz ohne, sprich komplex, speziell wenn das Protokoll nicht in eine URL ähnliche Form gebracht werden kann. Bei der Implementation eigener Protokolle können auch Fehler passieren.

Eine Alternative zu Sockets sind die Remote Procedure Calls (RPCs), welche die Kommunikationsschnittstelle abstrahieren und an Stelle von Sockets Prozeduraufrufe verwenden. In diesem Falle werden die Parameter für den Prozeduraufruf wie bei einem lokalen Prozeduraufruf behandelt. Die Applikation sieht sozusagen ein lokales Bild der entfernten Prozedur, obschon in Wirklichkeit die Parameter zu einer entfernten Prozedur gesendet werden. Die Parameter und die Rückgabewerte werden in einem neutralen Format dargestellt : XDR, external data representation.

RPC wurde zu einem Zeitpunkt geschaffen, als verteile Objektsysteme noch nicht das gängige Paradigma waren. RPC verwendet Prozeduraufrufe, nicht Objekte, welche auch noch in unterschiedlichen Adressräumen "leben". Damit entfernte Objekte sinnvoll zusammen arbeiten können, benötigt man zusätzliche "semantische" Modelle, also Modelle, die lokale und entfernte Objekte zusammen arbeiten lassen, kurz gesagt, eine Möglichkeit entfernte Methoden, Methoden entfernter Objekte einzusetzen : RMI

In einem solchen System ist es Aufgabe eines lokalen Subsystems (Stub) dafür zu sorgen, dass die entfernten Methoden korrekt aufgerufen werden, Parameter überprüft werden und allfällige nötige Parameter- oder Rückgabewerte-Konversionen vornehmen.

Die Java Remote Methode Invocation Spezifikation beschreibt die Architektur und Schnittstellen von RMI.

RMI wurde im Java Umfeld entwickelt. Dies impliziert, dass sowohl entfernte als auch lokale Objekte in einer Java Virtual Machine gespeichert sind und agieren.

1.2. System Ziele für RMI

Die Ziele für die Unterstützung verteilter Systeme in Java sind:

- Unterstützung von entfernten Methodenaufrufen, für Objekte in unterschiedlichen virtuellen Maschinen.
- Unterstützung von Callbacks vom Server zum Client (zum Beispiel einem Applet)
- Integration des verteilten Objektmodells in die Java Programmiersprache auf eine möglichst natürliche Art und Weise, also ohne Änderungen am Java Objekt Modell.
- allfällige Unterschiede zwischen dem verteilten Objektmodell und dem lokalen Java Objektmodell sollten transparent sein.
- das Schreiben von verteilten Applikationen soll so einfach wie möglich sein
- die Typenprüfung des Java Laufzeitsystems sollen erhalten bleiben
- unterschiedliche Mechanismen, wie die Aktivierung schlafender Objekt sollen möglich sein.
- die Java Mechanismen, wie Security Manager und Class Loader und Garbage Collector sollen auch in verteilten Systemen einsetzbar sein.

Zusammengefasst soll RMI einfach und leicht in das bestehende Java Umfeld integrierbar sein.

Wir werden in den nächsten Kapiteln das verteilte Objektsystem besprechen und anschliessend die Client und die Server Seite (API) des JDK1.2+ beschreiben, wenn auch etwas in Kurzfassung.

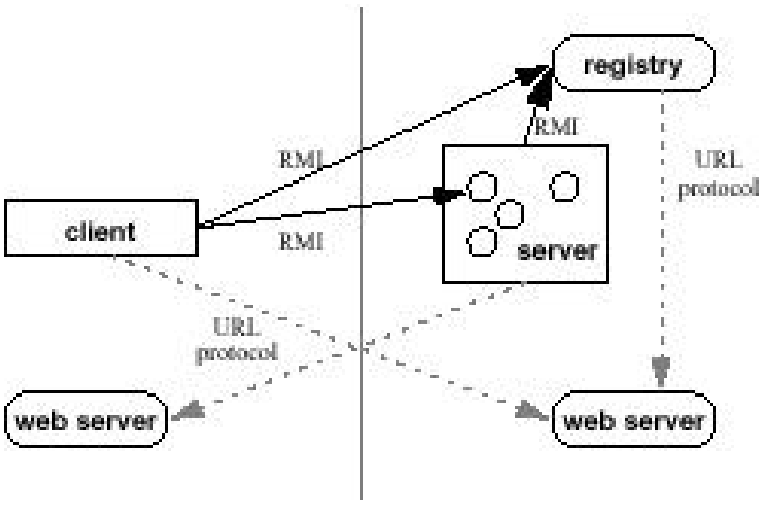
1.3. Verteiltes Objektmodell gemäss Java 2

1.3.1. Verteilte Objekt Applikationen

RMI Applikationen bestehen oft aus zwei separaten Programmen : einem Server und einem Client. Eine typische Serverapplikation kreiert eine Anzahl (aus Sicht des Clients entfernte, Server-) Objekte, erstellt Referenzen zu diesen Objekten (damit sie ansprechbar und eindeutig identifizierbar werden) und wartet dann darauf, dass eine Clientapplikation Methoden dieser entfernten Objekte aufruft. Eine typische Clientapplikation bestimmt als erstes eine Referenz zu einem oder mehreren entfernten Objekten im Server und ruft dann Methoden dieser Objekte auf.
RMI stellt Mechanismen zur Verfügung, mit deren Hilfe Client und Server miteinander kommunizieren können, in beiden Richtungen. Solche Applikationen bezeichnet man als verteilte objektorientierte Applikationen.

Verteilte Objektapplikationen müssen in der Lage sein:

- entfernte Objekte zu lokalisieren, zu finden
  Applikationen können zwei Mechanismen anwenden, um eine Referenz auf ein entferntes Objekt zu erhalten:
  1) eine Applikation kann ihre entfernten Objekte registrieren, mit Hilfe der `remiregistry` von RMI, mit deren Hilfe Namen in eine Liste eingetragen werden.
  oder
  2) die Applikation kann entfernte Objektreferenzen als Bestandteil seiner Operationen abfragen und abliefern
- mit entfernten Objekten zu kommunizieren.
  Die Details der Kommunikation zwischen den entfernten Objekten wird durch RMI behandelt. Der Aufruf einer entfernten Methode (der Methode eines entfernten Objekts) geschieht wie der Aufruf einer lokalen Methode.
- Klassen Bytecode für Objekte, die als Parameter übergeben oder empfangen werden zu laden.
  Da RMI dem Aufrufer einer entfernten Methode gestattet, Java Objekte als Parameter zu übergeben, stellt RMI auch die nötigen Mechanismen zur Verfügung, um sowohl Objektcode als auch Daten zu laden, zu übermitteln und zu schreiben.



Die Abbildug, aus der RMI Spezifikation (dies hier ist weitestgehend der Versuch diese in die Deutsche Sprache zu übersetzen) zeigt eine RMI verteilte Applikation, welche die Registry benutzt, um die Referenz ("Adresse") des entfernten Objekts zu erhalten.

Der Server ruft die Registry auf, um dem entfernten Objekt einen Namen zuzuordnen.
Der Client schaut das entfernte Objekt nach, indem es die Registry mit dem Objektnamen anfragt.

Zusätzlich wird noch ein Webserver verwendet, um Java Bytecode vom Server zum Client zu laden. RMI kann Klassen mit Hilfe des URL Protokolls (zum Beispiel HTTP, FTP, ...) laden, sofern dies vom aktuellen Java System unterstützt wird.

1.3.2. Begriffsbestimmungen

Im Java "Verteilten Objektmodell", ist ein *entferntes (remote) Objekt* ein Objekt, dessen Methoden von einer andern Java Virtuellen Maschine aus, die sich unter Umständen auch noch auf einem andern Rechner befindet, aufgerufen werden können. Ein Objekt dieses Typus wird mit Hilfe eines oder mehrerer *remote Interfaces* beschrieben. Diese Schnittstellen sind Java Schnittstellen, welche die Methoden eines entfernten Objekts beschreiben.

Remote Method Invocation (RMI) beschreibt die Aktion des Aufrufs einer Schnittstelle eines entfernten Objekts, dessen Methoden. Wichtig ist, dass der Aufruf einer entfernten Methode genau gleich wie ein lokaler Methodenaufruf erfolgt, also die selbe Syntax verwendet, wie die Methodenaufrufe bei lokalen Objekten.

1.3.3. Gegenüberstellung der Verteilten und Nichtverteilten Modelle

In Java sind die zwei Objektmodelle (lokal und verteilt) ähnlich, in folgendem Sinne:

- eine Referenz auf ein entferntes Objekt kann als Argument oder auch als Rückgabewert in Methodenaufrufen (lokal oder remote) verwendet werden.
- entfernte Objekte können umgewandelt werden (gecastet) gemäss der in Java implementierten Regeln für Umwandlungen.
- die Java Methode `instanceof` kann eingesetzt werden, um zu prüfen, ob es sich bei einem Objekt um ein lokales oder ein entferntes Objekt handelt.

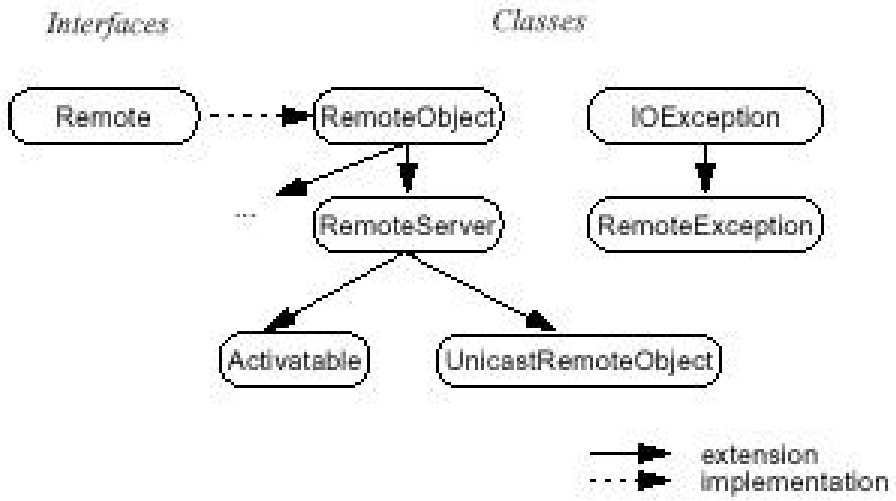
Das Java Verteilte Objektmodell unterscheidet sich vom Java Objektmodell in folgenden Aspekten:

- ein Client eines entfernten Objektes interagiert mit remote Interfaces, nie mit den Implementationsklassen dieser Schnittstellen.
- Lokale Argumente und Rückgabewerte werden als Kopien, nicht als Referenzen, an entfernte Methoden übergeben. Der Grund ist der, dass eine Referenz nur in einer lokalen Maschine Sinn macht. Die Adressräume einer entfernten JVM kann ganz anders aussehen, als die lokale JVM.
- entfernte Objekte werden als Referenzen, nicht als Objekte übergeben.
- einige der in `java.lang.Object` definierten Konstrukte müssen speziell angepasst werden.
- da die Fehleranfälligkeit verteilter Objektsysteme wesentlich höher ist als bei lokalen Objektsystemen, müssen Clients mit dieser zusätzlichen Komplexität fertig werden.

1.3.4. Übersicht über RMI Interfaces und Klassen

Die Schnittstellen und Klassen, welche für die Spezifikation des remote Verhaltens des RMI Systems verantwortlich sind, werden in der `java.rmi` Pakethierarchie definiert.

Die folgende Skizze zeigt Beziehungen zwischen einigen dieser Schnittstellen und Klassen:



1.3.4.1. Das java.rmi.Remote Interface

In RMI ist ein *remote* Interface eine Schnittstelle, welche von einer entfernten Java Virtuellen Maschine aufgerufen werden kann. Eine remote Schnittstelle muss folgende Anforderungen erfüllen:

- ein remote Interface muss mindestens entweder direkt oder indirekt die Schnittstelle `java.rmi.Remote` erweitern
- jede Methodendeklaration in einer remote Schnittstelle muss folgende Anforderungen an eine *remote Methodendeklaration* erfüllen:

1. eine remote Methode muss die Ausnahme `java.rmi.RemoteException` (oder eine ihrer Superklassen, wie `java.io.IOException` oder `java.lang.Exception`) neben eigenen applikationsspezifischen Ausnahmen in der `throw` Anweisung berücksichtigen.

2. in einer remote Methodendeklaration muss ein remote Objekt, welches als Parameter oder Rückgabewert auftritt (entweder direkt in der Parameterliste oder eingebettet in einem andern Objekt deklariert) als remote Interface, nicht als Implementationsklasse einer Schnittstelle definiert sein.

Das Interface `java.rmi.Remote` ist eine Platzhalter Schnittstelle, welche keine Methoden definiert:

```
public interface Remote {}
```

Ein Interface muss *mindestens* die Schnittstelle `java.rmi.Remote` (oder ein anderes Interface, welches `java.rmi.Remote` erweitert). Eine remote Schnittstelle kann unter bestimmten Bedingungen ein nicht entferntes Interface erweitern, unter folgenden Bedingungen:

- ein remote Interface kann ein anderes nicht remote Interface erweitern, sofern alle Methoden des erweiterten Interfaces die Anforderungen an eine remote Methodendeklaration erfüllt.

Zum Beispiel:
die folgende Bankkonto Schnittstelle definiert eine remote Schnittstelle für den Zugriff auf das Bankonto. Die Schnittstelle enthält Methoden, Geld einzuzahlen, abzuheben oder den Kontostand abzufragen.

```
public interface BankAccount extends java.rmi.Remote {
public void deposit(float amount)
throws java.rmi.RemoteException;
public void withdraw(float amount)
throws OverdrawnException, java.rmi.RemoteException;
public float getBalance()
throws java.rmi.RemoteException;}
```

Das nächste Beispiel zeigt ein gültiges remote Interface `Beta`, welches ein normales, lokales Interface `Alpha` erweitert. wobei `Alpha` und `Beta` remote Methoden besitzen und `Beta` `java.rmi.Remote` erweitert:

```
//ALPHA
public interface Alpha {
public final String okay = "auch Konstanten sind zugelassen";
public Object eineMethode(Object obj)
      throws java.rmi.RemoteExcepti on;
public void bar() throws java.io.IOException;
public int baz() throws java.lang.Exception;
}
//BETA
public interface Beta extends Alpha, java.rmi.Remote {
public void ping() throws java.rmi.RemoteException;}
```

1.3.4.2. Die RemoteException Klasse

Die `java.rmi.RemoteException` Klasse ist die Oberklasse der Ausnahmen, welche vom RMI Laufzeitsystem geworfen werden können. Um robuste RMI Applikationen zu erzeugen, muss jede remote Methode `java.rmi.RemoteException` (oder eine ihrer Oberklassen, wie zum Beispiel `java.io.IOException` oder `java.lang.Exception`) einbauen (... `throws` ...).

Die Ausnahme `java.rmi.RemoteException` wird immer dann geworfen, wenn eine entfernter Methodenaufruf fehlschlägt, aus irgend einem Grunde. Gründe für ein solches Fehlschlagen können sein:

- Kommunikationsfehler (der entfernte Server kann nicht erreicht werden, oder er verweigert den Zugriff; die Verbindung wird vom Server abgebaut, und viele andere Gründe)
- Fehler beim Verschlüsseln oder Entschlüsseln der Parameter und übermittelnten / zu übermittelnden Objekte (mashalling / unmarshalling).
- Protokollfehler

Die Klasse `RemoteException` ist eine Ausnahme, die von den Beteiligten, also nicht dem Laufzeitsystem geprüft wird (`RuntimeException`).

1.3.4.3. Die RemoteObject Klasse und ihre Unterklassen

RMI Serverfunktionen werden von `java.rmi.server.RemoteObject` und den Subklassen, `java.rmi.server.RemoteServer` und `java.rmi.server.UnicastRemoteObject` und `java.rmi.activation.Activatable` zur Verfügung gestellt.

- Die Klasse `java.rmi.server.RemoteObject` liefert Implementationen für die `java.lang.Object` Methoden `hashCode, equals` und `toString` für remote Objekte..
- Die Methoden, welche benötigt werden, um entfernte Objekte zu kreieren und zu exportieren (und damit entfernten Clients verfügbar zu machen), werden von den Klassen `UnicastRemoteObject` und `Activatable` zur Verfügung gestellt. Die Subklasse identifiziert die Semantik der remote Referenzen : handelt es sich um ein einfaches remote Objekt oder ist dieses aktivierbar und damit startbar, selbst wenn der Server das Objekt zur Zeit nicht aktiviert hat.
- Die `java.rmi.server.UnicastRemoteObject` Klasse definiert ein Singelton (Unicast) remote Objekt, dessen Referenzen nur dann gültig sind, wenn der Serverprozess aktiv, am Leben ist.
- Die Klasse `java.rmi.activation.Activatable` ist eine abstrakte Klasse, die ein aktivierbares remote Objekt definiert, also ein Objekt, welches gestartet wird, falls seine remote Methoden aufgerufen werden und anschliessend, falls nötig, wieder herunter gefahren werden.

1.3.5. Implementation eines Remote Interfaces

Generelle Regel zum Implementieren eines remote Interfaces:

- die Klasse erweitert *in der Regel* `java.rmi.server.UnicastRemoteObject`. Damit erbt das Objekt das (remote) Verhalten, welches durch die Klassen `java.rmi.server.RemoteObject` und `java.rmi.server.RemoteServer` vorgegeben wird.
- die Klasse kann eine beliebige Anzahl remote Interfaces implementieren
- die Klasse kann eine andere remote Implementationsklasse erweitern.
- die Klasse kann Methoden definieren, welche nicht im remote Interface beschrieben sind. Diese Methoden sind aber lediglich lokal, also nicht remote verfügbar.

Im folgenden Beispiel implementiert die Klasse `BankKontoImpl` das `BankKonto` remote Interface und erweitert die `java.rmi.server.UnicastRemoteObject` Klasse:

```
package meineBank;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class BankKontoImpl
            extends UnicastRemoteObject
            implements BankKonto {
      // Datenfelder
      private float balance = 0.0;
      // Konstruktor
      public BankKontoImpl(float initialerKontostand)
                  throws RemoteException{
            kontostand = initialerKontostand;
      }
      // Methoden
      public void einzahlen(float betrag) throws RemoteException {
```

```
      ...
      }
      public void abheben(float betrag) throws
            KontoUeberzogenException,
            RemoteException {
      ...
      }
      public float abfragenKontostand() throws RemoteException {
      ...
      }
}
```

Zu beachten ist, wenn auch eigentlich selbstverständlich, dass eine Klasse, welche ein remote Interface implementiert neben `java.rmi.server.UnicastRemoteObject` auch weitere Klassen erweitern und Schnittstellen implementieren kann.
Die Implementationsklasse muss dann allerdings die Verantwortung für das Exportieren der Objekte übernehmen (das geschieht normalerweise mit Hilfe des `UnicastRemoteObject` Konstruktors). Zudem muss die korrekte Semantik für die (remote) Methoden `hashCode`, `equals` und `toString`, welche von der `java.lang.Object` Klasse geerbt werden.

1.3.6. Parameterübergabe bei entfernten Methodenaufrufen (Remote Method Invocation)

Argumente für oder Rückgabewerte von remote Objekten (beziehungsweise präziser remote Methodenaufrufen) können alle gültigen Java Objekte sein, sofern sie ***serialisierbar*** sind. Dies beinhaltet die Java Basisdatentypen (primitive Datentypen), remote Java Objekte und nicht-remote Objekte, welche `java.io.Serializable` implementieren.
Die Objektserialisierung wird in der “Object Serialization Specification” beschrieben. Ein Beispiel finden Sie im Anhang. Klassen / Objekte (Parameter oder Rückgabewerte) , die lokal nicht verfügbar sind, werden vom RMI System dynamisch heruntergeladen (vom Client zum Server; vom Server zum Client). Wir kommen auf den Mechanismus des "Dynamic Class Loading" später nochmals zurück und werden uns dann im Detail damit befassen.

1.3.6.1. Übergabe von nichtentfernten Objekten

Ein nichtentferntes Objekt, welches als Parameter einer remote Methode verwendet wird, oder Rückgabe einer remote Methode ist, wird als ***Kopie*** übergeben. Das Objekt wird mit Hilfe der Java Objekt Serialisierung serialisiert.

Das Objekt, welches als Argument verwendet wird, wird also als erstes kopiert und dann serialisiert.

Wenn ein Objekt von einer remote Methode zurück gegeben wird, wird in der aufrufenden Virtuellen Maschine ein neues Objekt kreiert.

1.3.6.2. Übergabe von remote Objekten

Falls ein remote Objekt als Parameter auftritt oder als Rückgabewert einer remote Methode zurück gegeben wird, wird der Stub des remote Objekts übergeben. Ein remote Objekt, welches als Parameter verwendet wird, kann lediglich remote Interfaces implementieren.

1.3.6.3. Referentielle Integrität

Fall zwei Referenzen **eines** Objekts von einer VM zu einer andern VM gesendet werden (als Parameter oder Rückgabewerte von remote Methoden) beziehen sich diese zwei Referenzen auf ein und dasselbe Objekt der sendenden und der empfangenden VM.

Allgemeiner:
das RMI System sorgt für die referentielle Integrität der Objekte, die als Parameter oder Rückgabewerte verwendet werden.

1.3.6.4. Klassenbezeichnung

Wann immer ein Objekt von einer VM zu einer andern VM gesendet wird, unterhält das RMI System die Klassenbeschreibung. Dies ist eine Anforderung an das RMI System bzw. Klassen, die in einem RMI System verwendet werden.

1.3.6.5. Parameterübergabe

Parameter in einem RMI Aufruf werden in einen Objektstrom geschrieben. Dieser Objektstrom ist eine Unterklasse von `java.io.ObjectOutputStream` und wird eingesetzt, um die Parameter zu serialisieren und an die entfernte VM zu übermitteln. Die `ObjectOutputStream` Subklasse überschreibt die `replaceObject` Methode, um alle remote Objekte durch ihre entsprechenden Stubklassen zu ersetzen.

Parameter, welche Objekte darstellen, werden mit Hilfe der `writeObject` Methode des `ObjectOutputStream` in den Objektstrom geschrieben.

Der `ObjectOutputStream` ruft die `replaceObject` Methode für jedes Objekt auf, welches mit Hilfe der `writeObject` Methode in den Objektstrom geschrieben wurden. Die `replaceObject` Methode der RMI Unterklasse von `ObjectOutputStream` liefert folgendes:

- falls das Objekt, welches als Argument von `replaceObject` auftritt, eine Instanz von `java.rmi.Remote` ist,
  dann liefert die Methode replaceObject den Stub des remote Objekts.
  Einen Stub für ein remote Objekt erhält man mit Hilfe der Methode `java.rmi.server.RemoteObject.toStub`.
- falls das Objekt, welches als Argument von `replaceObject` auftritt, **keine** Instanz von `java.rmi.Remote` ist,
  dann wird einfach das Objekt aus dem Objektstrom rekonstruiert.

RMI's Subklasse von `ObjectOutputStream` implementiert auch die annotateClass Methode. Wir kommen im Abschnitt "Dynamisches Laden von Klassen" auf dieses Thema zurück.

Da Parameter in einen einzigen ObjectOutputStream gescgrieben werden, referenzieren Aufrufe auf beiden Seiten des Objektstromes (Sender und Empfänger) die selbe Kopie des Objektes.

Bei `writeReplace` beim Schreiben und `readResolve` beim Lesen von Objekten sorgt der RMI Mashalling und UnMarshalling Mechanismus für das korrekte Verhalten.

1.3.7. Lokalisierung von Remote Objekten

Mit Hilfe eines einfachen Namensserver werden Namensreferenzen zu remote Objekten verwaltet. Eine remote Objektreferenz kann mit Hilfe der URL-basierten Methoden der Klasse `java.rmi.Naming` gespeichert werden.

Damit ein Client eine Methode eines remote Objekts aufrufen kann, muss der Client zuerst eine Referenz auf dieses Objekt erhalten. Eine Referenz zu einem remote Objekt erhält man normalerweise als Parameter oder Rückgabewert eines Methodenaufrufs.

Das RMI System stellt einem einfachen "bootstrap" Namensserver zur Verfügung, mit dessen Hilfe remote Objekte eines gegebenen Hosts erhalten werden können. Die java.rmi.Naming Klasse stellt Uniform Resource Locator (URL) basierte Methoden zur Verfügung, mit denen <Objekt, Namen>s Paare nachgeschlagen (lookup), gebunden (bind) oder gelöscht (unbind), neu gebunden (rebund) und aufgelistet (list) werden können, auf einem bestimmten Host und an einem bestimmten Port.

1.3.8. Einführendes Beispiel

Im Folgenden wollen wir ein einfaches RMI Programmbeispiel implementieren. Dabei handelt es sich um ein "Hello World" RMI Programm. Anschliessend werden wir komplexere Beispiele (Chat, Terminkalender) kennen lernen.

Weil wir bisher wenig Applets verwendet haben, werden wir das Beispiel mit Hilfe eines Applets realisieren, inklusive HTML Tags und allem Drum und Dran.

Das Applet wird von einem Server geladen und ruft dann mit Hilfe von RMI eine Methode auf dem Server auf, die das String Objekt "Hello World" zurück liefert. Diese Zeichenkette wird auf dem Client, im Applet der HTML Seite angezeigt.

Das "Projekt" realisieren wir in drei Schritten:

- zuerst schreiben wir die HTML Seite und die Java Programme.
- dann übersetzen wir die Java Programme und "verteilen" Class und HTML Dateien.
- schliesslich starten wir die Remote Objekt Registry, den Server und das Applet.

Soweit der Plan; fangen wir also mit dem ersten Schritt an.

1.3.8.1. Schreiben der HTML und Java Quelldateien

Insgesamt vier Quelldateien werden benötigt:

1. das Java remote Interface
2. das Java remote Objekt (der Server), welches das remote Interface implementiert
3. das Java Applet (der Client), welcher die Servermethode aufruft
4. die HTML Seite, welche das Applet enthält

Da die Java Programmiersprache eine Beziehung zwischen dem Paketnamen und Verzeichnissen herstellt, muss man sich vor dem Schreiben des Programms klar werden, wie das Paket heissen soll und wo die Dateien gespeichert werden.

Für das HelloWorld Programm wählen wir den Paketnamen HelloWorld, im entsprechenden Programmverzeichnis zu diesem Kapitel.

1.3.8.2. Definition eines Remote Interfaces

Da bei einem remote Methodenaufruf viele Fehler geschehen können, zum Beispiel wegen Netzwerkfehlern, Kommunikationsfehler, Serverfehlern oder Cllientfehlern. Damit eine remote Methode implementiert werden kann, muss diese folgenden Charakteristiken haben:

- das remote Interface muss `public` sein. Sonst wird der Client einen Fehler produzieren, sobald er versucht ein remote Objekt zu laden, um das remote Interface zu implementieren
- das remote Interface muss das Interface `java.rmi.Remote` erweitern
- jede Methode muss die java.rmi.RemoteException deklarieren, zusätzlich zu applikationsspezifischen Ausnahmen.
- ein remote Objekt, welches als Argument oder Rückgabewert (direkt oder eingebettet in ein anderes lokales Objekt) auftritt, muss als remote Interface deklariert werden, nicht als Implementationsklasse.

Wie sieht das Interface unseres HelloWorld Programms aus?

Das Interface enthält lediglich eine einzige Methode, `sayHello`, welche die Zeichenkette zurück liefert:

```
package HelloWorld;
public interface Hello extends java.rmi.Remote {
      String sayHello() throws java.rmi.RemoteException;
}
```

1.3.8.3. Schreiben einer Implementationsklasse

Ein remote Objekt wird mit Hilfe einer Klasse beschrieben, welche eines oder mehrere remote Interfaces implementiert. Die Implementationsklasse muss:

1. das remote Interface, welches implementiert werden soll, spezifizieren.
2. den Konstruktor für das remote Objekt definieren
3. eine Implementation der Methoden, die remote eingesetzt werden sollen, spezifizieren.
4. einen Security Manager kreieren und installieren
5. eine oder mehrere Instanzen eines remote Objektes kreieren
6. mindestens eines der remote Objekte mit Hilfe der RMI remote Objektregistry registrieren, damit ein Bootstrappen der Kommunikation möglich wird.

In unserem Beispiel sieht die Implementation für unser HelloImpl.java des Servers folgendermassen aus:

```
package HelloWorld;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class HelloImpl extends UnicastRemoteObject implements Hello {
  private String name;
  public HelloImpl(String s) throws RemoteException {
    super();
    name = s;
  }
  public String sayHello() throws RemoteException {
    return "Hello World!";
  }
  public static void main(String args[]) {
  // kreieren und installieren eines Security Managers
    System.setSecurityManager(new RMISecurityManager());
    try {
      HelloImpl obj = new HelloImpl("HelloServer");
      Naming.rebind("//localhost/HelloServer", obj);
      System.out.println("HelloServer in die Registry eingetragen!");
    } catch (Exception e) {
      System.out.println("HelloImpl err: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

1. Implementieren eines remote Interfaces
   Die Implementationsklasse für unser HelloWorld Beispiel heisst `HelloImpl.java`.
   Eine Implementationsklasse spezifiziert das / die remote Interface(s), welches implementiert werden soll. Optional kann der Server, welcher erweitert wird, angegeben werden. In unserem Falle ist dies `java.rmi.server.UnicastRemoteObject`.

   Die Deklaration der HelloImpl Klasse sieht wiefolgt aus:

```
public class HelloImpl extends UnicastRemoteObject
                            implements Hello
```

Die Tatsache, dass unsere Klasse UnicastRemoteObject erweitert, zeigt an, dass unsere Klasse eingesetzt werden soll, um ein einfaches (nicht repliziertes) remote Objekt zu kreieren und dass RMI die Standard Socket-basierte Transportmethoden für die Kommunikation verwendet.

Falls ein entferntes Objekt einer nichtentfernten Klasse erweitert werden soll, muss dieses remote Objekt explizit mit Hilfe der Methode `UnicastRemoteObject.exportObject` exportiert werden.

2. Definition des Konstruktors für das remote Objekt
   Der Konstruktor für eine entfernte Klasse unterscheidet sich nicht vom Konstruktor einer nichtentfernten Klasse : die Variablen jeder neu kreierten Instanz der Klasse werden initialisiert.

```
public HelloImpl(String s) throws RemoteException {
    super();
    name = s;
}
```

   Was geschieht im Einzelnen?
   - die `super` Methode ruft den argumetlosen Konstruktor der `java.rmi.server.UnicastRemoteObject` Klasse auf. Diese“exportiert” das remote Objekt und hört auf ankommende Anfragen, an einem anonymen Port.
   - der Konstruktor muss die Ausnahme `java.rmi.RemoteException` werfen können, weil RMI's Versuch ein remote Objekt zu exportieren, schief gehen kann, falls zum Beispiel die Kommunikation anbricht.

   Der Aufruf der `super` Methode würde auch implizit geschehen, also ohne expliziten Aufruf. Die Schreibweise zeigt aber besser, dass zuerst das Objekt der Oberklasse angelegt wird und erst dann daraus das gewünschte Objekt gebaut wird.

3. Jede remote Methode muss implementiert werden.
   Die Implementationsklasse für ein remote Objekt enthält Programmcode, welcher jede der remote Methoden, welche im remote Interface spezifiziert wurde, implementiert.

   In unserem Beispiel:

```
public String sayHello() throws RemoteException {
    return "Hello World!";
}
```

   Argumente und Rückgabewerte von remote Methoden können von irgend einem Java Typus sein, also auch Objekte, sofern diese Objekte das Interface `java.io.Serializable` implementieren (die Objekte müssen ja unter Umständen von einer JVM zur andern geschickt werden).
   Die meisten Basisklassen in Java, die in `java.lang` und `java.util` beschrieben sind, implementieren das `Serializable` Interface.

   - lokale Objekte werden als Kopien übergeben. Per Default werden nur nichtstatische und nichttransiente Datenfelder kopiert.
   - remote Objekte werden als Referenzen übergeben. Eine Referenz auf ein remote Objekt

ist in Wirklichkeit eine Referenz auf einen Stub, welcher einen Proxy für das entfernte Objekt darstellt. Stubs werden wir später noch im Detail kennen lernen.

Bemerkung:
Eine Klasse kann Methoden deklarieren und implementieren, die in der Spezifikation des remote Interfaces nicht aufgeführt sind. Diese Methoden können allerdings lediglich in der lokalen Virtuellen Maschine ausgeführt werden, also nicht remote.

4. kreieren und installieren eines Security Managers
   Die `main` Methode des Dienstes muss als erstes einen Security Manager kreieren und installieren:
   - entweder den `RMISecurityManager`
   - oder einen selbst definierten.

   In unserem Beispiel:
   ```
   System.setSecurityManager(new RMISecurityManager());
   ```

   Ein Security Manager muss geladen werden, um zu garantieren, dass keine der Klassen "sensitive" Operationen ausführen kann.
   Falls kein Security Manager spezifiziert wird, können RMI Klassen nicht geladen werden, zumindest in einem Netzwerk; lokal kann dies anders aussehen.

5. Kreieren einer oder mehrerer Instanzen eines remote Objektes
   Die `main` Methode des Services muss eine oder mehrere Instanzen des remote Objekts, welche den Dienst anbieten, kreieren

   In unserem Beispiel:
   ```
   HelloImpl obj = new HelloImpl("HelloServer");
   ```

   Der Konstruktor "exportiert" das remote Objekt. Das heisst, dass das remote (Server-) Objekt, nachdem es kreiert wurde, bereit ist, auf ankommende Anrufe zu "hören".

6. Registrieren eines remote Objekts
   Damit ein Anrufer (Client, Peer oder Applet) eine Methode eines remote Objektes aufrufen kann, muss der Anrufer zuerst eine Referenz auf das remote Objekt erhalten. Meistens geschieht dies mit Hilfe eines Methodenaufrufes : die Referenz wird als Rückgabeparameter geliefert.

   Damit das System gestartet werden kann, benötigen wir einen Bootstrapping Mechanismus. RMI verwendet eine URL-basierte Registry, mit deren Hilfe eine URL der Form `//host/objektname` auf ein entferntes Objekt verwendet werden kann. `objektname` ist dabei eine einfache Zeichenkette.

   Nachdem ein remote Objekt auf dem Server registriert ist, können Anrufer das Objekt beim Namen ansprechen, eine remote Objektreferenz erhalten und dann eine remote Methode aufrufen.

   In unserem Beispiel:
   ```
   Naming.rebind("//localhost/HelloServer", obj);
   ```

   Bemerkungen:

- falls der Host fehlt, wird `localhost` angenommen.
- es muss kein Protokoll angegeben werden
- das RMI Laufzeitsystem ersetzt eine Referenz auf ein remote Objekt durch eine Referenz auf den Stub für dieses Objekt. Remote Implementationen verlassen die VM nie. Wenn ein Client einen Lookup durchführt, wird eine Referenz auf den Stub zurück gegeben.
- optional kann eine Portnummer angegeben werden, zum Beispiel
  `//meinHost:1121/HelloServer`
  Ohne Portangabe wird der Port **1099** verwendet. Falls ein anderer Port verwendet werden soll, muss der Port angegeben werden.

- aus Sicherheitsgründen kann eine Applikation nur an/von die/der Registry gebunden / entbunden werden, welche auf dem selben Host läuft, **wie der Server**.
  Dadurch kann verhindert werden, dass ein Eintragg in der Registry eines Servers überschrieben oder gelöscht werden kann. Lookup / Lesen ist von jedem Server möglich.

1.3.8.4. Schreiben eines Applets, welches den Remote Service verwendet

Das Applet unserer Applikation ruft den HelloServer auf, speziell dessen sayHello Methode und erhält damit die Zeichenkette "Hello World!" zurück. Diese Zeichenkette wird auf der HTML Seite im Applet angezeigt.

In unserem Beispiel:

```
package HelloWorld;

import java.awt.*;
import java.rmi.*;

public class HelloApplet extends java.applet.Applet {
  String message = "";
  public void init() {
    try {
      Hello obj = (Hello)Naming.lookup("//" +
      getCodeBase().getHost() + "/HelloServer");
      message = obj.sayHello();
    } catch (Exception e) {
      System.err.println("HelloApplet Exception: " +
      e.getMessage());
      e.printStackTrace();
    }
  }
  public void paint(Graphics g) {
    g.drawString(message, 25, 50);
  }
}
```

1. Das Applet holt sich als erstes eine Referenz zum "HelloServer" aus der Server Registry

   ```
   Hello obj = (Hello)Naming.lookup("//" +
                       getCodeBase().getHost() + "/HelloServer");
   ```

   und konstruiert die URL mit Hilfe der `getCodeBase()` und der `getHost()` Methode.

2. nun ruft das Applet die remote Methode des HelloServers auf :

   ```
   message = obj.sayHello();
   ```

   und speichert den Rückgabewert ("Hello World!" oder was auch immer der Server sendet) in der Variable `message`.

3. das Applet ruft die paint Methode auf, um das Applet auf der Anzeige zu zeichnen, also die Zeichenkette "Hello World!" anzuzeigen.

Bemerkung:
Die konstruierte URL muss den Host spezifizieren. Sonst versucht das Applet lokal im Client die Namensauflösung durchzuführen. In diesem Falle wird der `AppletSecurityManager` eine Ausnahme werfen, da das Applet nicht auf das lokale System zugreifen kann und darf. Das Applet darf nur mit dem Applet Host kommunizieren.

1.3.8.5. Schreiben der Webseite, die das Applet enthält

Der HTML Code wird entweder von der Applet Entwicklungsumgebung generiert, oder er lässt sich einfach in einem Texteditor schreiben.

In unserem Beispiel:

```
<HTML>
<title>Hello World</title>
<center> <h1>Hello World</h1> </center>
Die Meldung vom Server lautet:
<p>
<applet codebase="../.." (sofern das Applet dort steht)
code="HelloWorld.HelloApplet"
width=500 height=120>
</applet>
</HTML>
```

Dabei gilt es folgendes zu beachten:

1. es wird ein HTTP Server benötigt, auf dem die Server Applikation läuft und von dem das Applet herunter geladen werden soll. Dabei wird das Verzeichnis, aus dem das Applet gelesen werden soll, mit Hilfe des Tags

   `CODEBASE = "."` (wo auch immer das Verzeichnis ist)

   angegeben.
   In diesem Beispiel handelt es sich um das aktuelle Verzeichnis, also das Verzeichnis aus dem die HTML Seite geladen wurde.

2. die Spezifikation des Appletcodes wird mit Hilfe des Tags

   `CODE = "HelloWorld.HelloApplet.class"` (mit Paketnamen)

   angegeben.

1.3.8.6. Übersetzen und verteilen der Class und HTML Dateien

Nun sind die Java Programme vollständig. Insgesamt haben wir also folgende Dateien:

- `Hello.java`, welches das remote Interface beschreibt
- `HelloImpl.java`, welches das remote Objekt implementiert, den Server für die Hello World Applikation.
- `HelloApplet.java`, unser Applet
- `HelloWorld.HelloApplet.html`, die Webseite für unser Applet.

Da alle obigen Java Quellen in der Entwicklunsumgebung erstellt wurden, ist es kein Problem gleich Class Dateien zu generieren.

Zudem müssen wir die bereits mehrfach erwähnten Stubs und Skeletons generieren. Dies geschieht mit Hilfe des `rmic` Compilers.

Der rmic Compiler kann im JBuilder aktiviert werden, indem man im Fenster links iben, dort wo die Java Dateien aufgelistet werden, mit der rechten Maustaste die Eigenschaften der Dateien geändert werden, konkret kann dort RMI aktiviert werden.

Abbildung 23-0-1 RMI Stub / Skeleton generieren

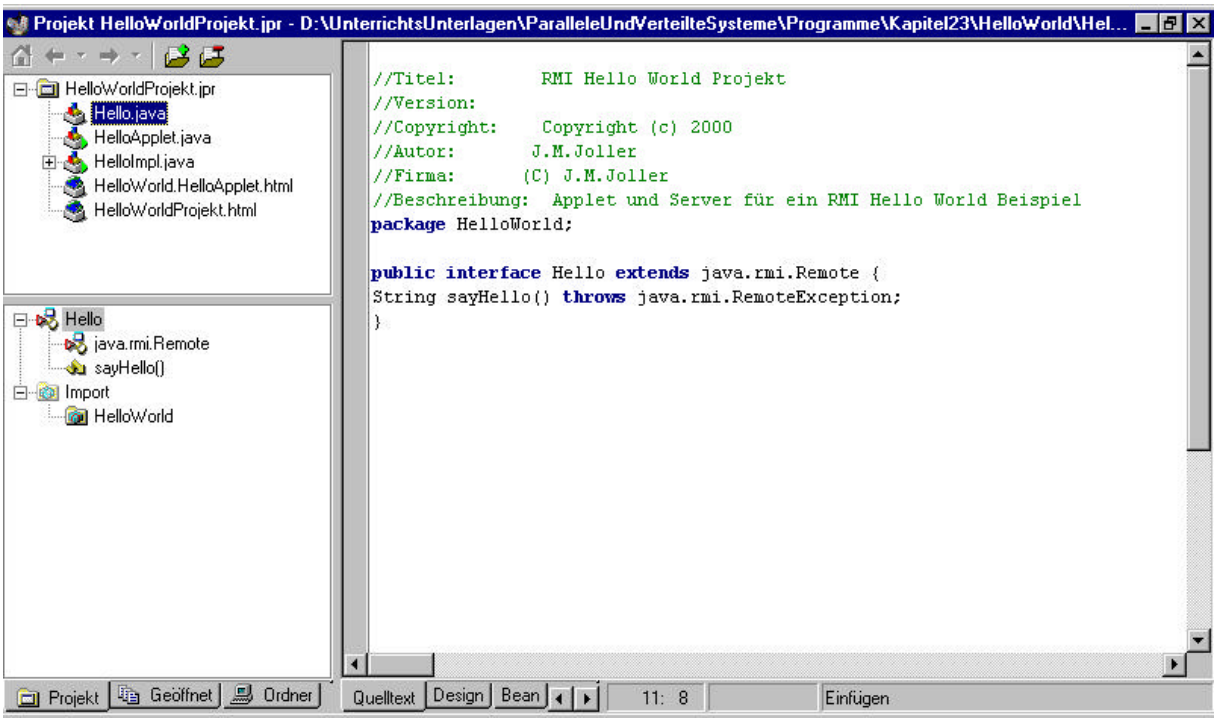


Abbildung 23-0-2 Dateifenster im JBuilder

Zur Wiederholung:

- ein Stub ist ein Proxy auf dem Client für remote Objekte. Dieser Proxy sendet RMI Anfragen an die serverseiten Skeletons. Das Skeleton leitet den Aufruf an die aktuelle Implementation des remote Objektes weiter.

Beim manuellen Aufruf des rmic muss das Ausgabeverzeichnis angegeben werden. Ein typischer Aufruf sieht in diesem Falle folgendermassen aus:

```
rmic -d . *.java
```

falls die resultierenden Class Dateien im aktuellen (".") Verzeichnis gespeichert werden sollen.

1.3.8.7. Übersetzen der Java Quellprogramme

Im JBuilder werden die Java Dateien wie üblich übersetzt. Falls keine Entwicklungsumgebung zur Verfügung steht, kann dies mit Hilfe vom JDK geschehen:

```
javac -d Hello.java HelloImpl.java HelloApplet.java
```

Falls die Dateien eine Paketspezifikation enthalten, werden Unterverzeichnisse angelegt.

1.3.8.8. Generieren von Stubs und Skeletons

Der Aufruf des `rmic` Compilers haben wir bereits oben gesehen. Das Ergebnis von rmic sind Stubs und Skeleton Dateien.

Im Falle des JBuilders , nachdem die Eigenschaften der Dateien gesetzt wurden, werden nicht nur Class Dateien, sondern auch noch Java Dateien generiert.

Daraus ist ersichtlich, dass HashCodes eingefügt wurden, mit denen effizient auf die spezifizierten Methoden zugegriffen werden kann.

- `HelloImpl_Stub.class`
- `HelloImpl_Skel.class`

Der Stub muss natürlich exakt die selben Methoden wie das remote Interface implementieren, sonst würde die Proxy Kommunikation nicht funktionieren.

Auf der Client Seite kann man alle Java spezifischen Konstrukte, wie Casting, Typenprüfung und ähnliches einsetzen. Auf der Server Seite kann man Polymorphismus, ... verwenden.

1.3.8.9. Verschieben der HTML Seite

Nun müssen wir die Pfade überprüfen! Im HTML Code müssen wir die Codebase Variable gemäss der aktuellen Situation auf unserem Server setzen.

1.3.8.10. Setzen der Pfade für die Laufzeit

Die Codebase und der CLASSPATH des Servers müssen definiert sein, bevor der Server gestartet werden kann.

1.3.8.11. Start der Objekt Registry, des Servers und des Applets

Falls alle Pfade korrekt gesetzt wurden, sollte unser System jetzt auf gestartet werden können.

1.3.8.11.1. Start der RMI Bootstrap Registry

Die RMI Registry dient als einfacher Bootstrap Namensserver, mit dessen Hilfe ein remote Client eine Referenz auf ein remote Objekt erhalten kann. Typischerweise wird wird die Registry lediglich zum Bootstrappen eingsetzt. Die weiteren Referenzen werden dann applikationsseitig erstellt.

Der Start der Registry geschieht mit Hilfe des `rmiregistry` Befehls, oder mit Hilfe von `start rmiregistry` oder innerhalb von JBuilder (Tools)

Falls ein Fehler den Start verhindert, kann es sein, dass der Port (**1099**) bereits besetzt ist. In diesem Falle kann durch Angabe des Ports ein anderer Port aktiviert werden:
`rmiregistry 2001` startet die Registry, aber an Port 2001.

Im Programm muss der Server entprechend angepasst werden:

```
Naming.rebind("//myhost:2001/HelloServer", obj);
```

Auch im Applet muss der Port angepasst werden:

```
<PARAM name="url" value="//meinHost:2001/HelloServer">
```

Die Registry bindet Stub und Skeleton. Falls diese verändert werden, muss die Registry neu gestartet werden

1.3.8.11.2. Start des Servers

Beim Starten des Servers muss die `java.rmi.server.codebase` Eigenschaft spezifiziert werden. Dadurch kann beim Herunterladen der Stubklasse zum Client die URL dem Client korrekt mitgeteilt werden:

```
java –Djava.rmi.server.codebase=http://meinHost/...Verzeichnis.../codebase/
      HelloWorld.HelloImpl
```

Die Option `-D` dient dem Setzen von "Properties" für die Java Virtual Machine.

Bemerkung – Zu beachten ist `/` in der `codebase` URL (am Schluss).

Eine Stub Klasse wird dynamisch in die VM des Clients geladen, sofern die Klasse lokal noch nicht vorhanden ist.

1.3.8.11.3. Start des Applets

Nachdem nun die Registry und der Server gestartet wurden, können wir uns das Applet anschauen:
entweder mit dem Appletviewer oder in einem Java fähigen Browser

`appletviewer` http://meinHost/~meinUsername/codebase/.../HelloWorld/HelloWorld.HelloApplet.html

Sie sollten dann ein Bild wie unten sehen:

Abbildung 23-3 HelloWorld RMI Applet

1.4. RMI - Übersicht

1.4.1. Stubs und Skeletons

RMI benutzt einen Standard Mechanismus, der bereits bei RPC eingeführt wurde, für die Kommunikation zwischen remote Objekten : *Stubs* und *Skeletons*.
Ein Stub für ein remote Objekt agiert als lokaler Repräsentant oder Proxy für das remote Objekt. Das aufrufende Programm verwendet eine Methode des lokalen Stubs, der verantwortlich für den Methodenaufruf des entfernten Objekt ist. In RMI implementiert ein Stub eines remote Objekts das selbe Interface, wie das remote Objekt implementiert.
Die Aufgaben des (dazugehörenden) Stubs sind:

- der Verbindungsaufbau zur remote VM, welche das remote Objekt enthält.
- *marshall* (umwandeln, schreiben und übermitteln) der Parameter zur remote VM
- warten auf das Ergebnis des Methodenaufrufs
- *unmarshall* (lesen und übermitteln) des Rückgabewertes oder einer Ausnahme
- abgeben des Rückgabewertes an den Aufrufer

Der Stub versteckt die Serialisierung der Parameter und die Kommunikation auf der Netzwerkebene, damit der Methodenaufruf für den Aufrufenden möglichst einfach gestaltet werden kann.

In der entfernten VM hat jedes remote Objekt ein ihm entsprechendes Skeleton, ausser in einer reinen JDK 1.2+ Umgebung, in der Skeletons nicht mehr benötigt werden.
Das Skeleton ist für die Weiterleitung, das *Dispatchen*, des Aufrufs an die aktuelle remote Objekt Implementation verantwortlich. Falls ein Skeleton einen Methodenaufruf empfängt, läuft im Einzelnen folgendes ab:

- die Parameter für die remote Methode werden entschlüsselt (*unmashalling*)
- die Methode der aktuellen Objektimplementation wird aufgerufen
- das Ergebnis (Rückgabewert oder Ausnahme) wird verschlüsselt und an den Anrufer zurück geschickt (geschrieben und übermittelt : *marshalling*)

Im Rahmen von JDK1.2+ wurde ein weiteres Stub Protokoll eingeführt, mit dessen Hilfe für Skeletons ab JDK 1.2 Skeletons überflüssig werden. Statt dessen wird generischer Code benutzt, mit dessen Hilfe die selben Aufgaben erledigt werden können, wie mit einem Skeleton der JDK Version 1.1 und älter.

Stubs und Skeletons werden mit Hilfe des `rmic` (remote method invocation compiler) generiert.

1.4.2. Einsatz von Threads in RMI

Eine Methode, welche vom RMI Laufzeitsystem an eine entfernte Objektimplementation weitergeleitet wird, kann muss aber nicht in einem separaten Thread ablaufen.
Das RMI Laufzeitsystem kümmert sich aber nicht um Threads.

Ein entfernter Methodenaufruf kann gleichzeitig von verschiedenen Clients abgesetzt werden.

Daher muss die *Implementation* des remote Objekts dafür sorgen, dass es Thread-sicher ist.

1.4.3. Garbage Collection von Remote Objekten

In einem verteilten System, genau wie in einem lokalen, ist es wünschenswert, alle remote Objekte, welche nicht länger von Clients referenziert werden, zu löschen. Dies sollte automatisch geschehen, damit sich der Programmierer nicht darum zu kümmern hat. RMI benutzt einen Referenzzähleralgorithmus, analog jenem von Modula-3, einer experimentellen Sprache des Digital Equipment Corporate Systems Research Centers (“Network Objects” von Birrell, Nelson, and Owicki, *Digital Equipmen Corporation Systems Research Center Technical Report 1*15, 1994).

Dazu verwendet das RMI Laufzeitsystem Referenzentabellen in jeder VM. Wann immer eine Referenz in einer JVM aktiviert wird, beispielsweise als Rückgabewert eines Methodenaufrufes, wird der Referenzzähler erhöht. Beim ersten Auftreten einer Referenz wird eine Referenzmeldung an den Server gesandt. Sobald Referenzen nicht mehr benötigt werden, beispielsweise weil sie auf `null` gesetzt wurden, wird der Referenzzähler um eins reduziert, zuerst in der lokalen VM. Wenn ein Objekt uberhaupt nicht mehr referenziert wird, sendet die lokale VM eine "unreferenziert" Meldung an den Server. Der Mechanismus ist in Wirklichkeit wesentlich feiner, vorallem um sicherzustellen, dass ein Objekt nicht zu früh eliminiert wird.

Falls ein entferntes Objekt weder lokal noch remote referenziert wird, spricht man von einer "schwachen" oder "weak" Referenz. Falls keine andern lokalen Referenzen auf das Objekt existieren, kann der Garbage Collector das Objekt entfernen. Der *verteilte* Garbage Collector (dgc : distributed garbage collector) arbeitet mit dem lokalen Garbage Collector zusammen.

Solange lokale Referenzen auf ein entferntes Objekt existieren, darf der Garbage Collector das Objekt nicht eliminieren können. Remote Objekte fügen neben dem Referenzzähler auch noch die Kennnung der VM hinzu, auf die das Objekt verbunden wurde. Das Dereferenzieren eines remote Objektes kann man auch mit Hilfe des Interfaces `java.rmi.server.Unreferenced` bewerkstelligen; diese Schnittstelle muss also implementiert werden, sofern man in der Lage sein will, remote Objekte zu dereferenzieren. Nach dem Aufruf der `unreferenced` Methode wird ein Objekt ein Kandidat für den Garbage Collector, aber nur, falls alle lokalen und entfernten Referenzen dereferenziert werden.

Weil in einem komplexen Netzwerk immer wieder Fehler passieren können und auf Grund von *Timeouts* Operationen ausgelöst werden können, ist es möglich, dass ein remote Objekt zu früh entfernt wird, zum Beispiel weil der Transportlayer annimmt, dass die Verbindung unterbrochen wurde.

Aus diesem Grunde kann für remote Referenzen **keine** referentielle Integrität garantiert werden, anders gesagt, eine remote Referenz kann unter Umständen ins Leere zeigen. Eine Verwendung solcher Referenzen führt dann zum Werfen einer `RemoteException,` welche in der Applikation behandelt werden muss.

1.4.4. Dynamisches Laden von Klassen

RMI gestattet irgendwelche Objekte als Parameter, Rückgabewerte und Ausnahmen einzusetzen, sofern sie serialisierbar sind. RMI setzt den Objektserialsierungs-Mechanismus ein, um Daten von einer VM zur andern zu transportieren und Aufrufe mit entsprechenden Lokalisierungsinformationen (auf welcher VM ist das Objekt verfügbar) zu versehen, damit die Klassendateien gefunden und geladen werden können.

Nachdem die Parameter und Rückgabewerte entschlüsselt (unmarshalled) wurden, werden die Objekte in der lokalen VM konstruiert. Dazu werden die Definitionen der Objekte benötigt. Dazu versucht der Unmashall Prozess zuerst Informationen in der lokalen Umgebung zu finden. RMI stellt zusätzlich einen Mechanismus zur Verfügung, dynamisch die Klassendefinitionen zu laden. Dies umfasst auch das Laden von Stub Information für bestimmte Objektimplementationsklassen.

Für das dynamische Laden von Klassen verwendet das RMI Laufzeitsystem spezielle Unterklassen der Objekt Eingabe- und Ausgabe-Ströme `java.io.ObjectOutputStream` und `java.io.ObjectInputStream.` Diese Unterklassen überschreiben die `annotateClass` Methode des `ObjectOutputStream` und der `resolveClass` Methode des `ObjectInputStream` und kommunizieren Informationen darüber, *wo die Klassenbeschreibungen im Objektstrom zu finden sind.*

Pro Klassenbeschreibung, die in den RMI (marshall) Strom geschrieben wird, fügt die `annotateClass` Methode das Ergebnis des Methodenaufrufs `java.rmi.server.RMIClassLoader.getClassAnnotation` für das Objekt hinzu. Diese Information kann `null` sein oder sie steht als `String` Objekt zur Verfügung, in dem der URL Pfad steht (eine URL pro Objekt / Klasse, mit einem Leerzeichen zwischen den einzelnen URLs), von wo die Definition der Klasse für die gegebene Klasse herunter geladen werden kann. Mit Hilfe der `resolveClass` Methode werden für die einzelnen Objekte die Klassenbeschreibung aus dem RMI kodierten Objektstrom gelesen.

Die `resolveClass` Methode wird aufgerufen, um das ganze Objekt aus dem RMI Strom zu lesen. Die `resolveClass` Methode ihrerseits ruft die Methode `RMIClassLoader.loadClass` Methode auf.

Wir werden diese Aufrufe noch genauer anschauen.

1.4.5. RMI Zugriffe durch eine Firewall via Proxies

Der RMI Transport Layer versucht normalerweise eine Socketverbindung zu den Hosts aufzubauen. Falls eine Firewall den direkten Zugriff nicht erlaubt, versucht RMI mit Hilfe eines HTTP basierten Mechanismus zu kommunizieren.

1.4.5.1. Wie wird ein RMI Call im HTTP Protokoll verpakt?

Um durch die Firewall zu kommunizieren bettet der RMI Transportlayer einen RMI Aufruf in das in der Regel Firewall-trusted HTTP Protokoll. Die RMI Daten werden als Rumpf eines HTTP `POST` Aufrufs nach aussen gesandt, die (RMI) Antwort steht dann einfach in der HTTP Antwort dieses Befehlaufrufs. Der eigentliche `POST` Aufruf geschieht auf zwei mögliche Arten:

1. falls der Firewall Proxy den HTTP Befehl direkt an einen beliebigen Port der Zielmaschine abliefert, dann wird der `POST` Befehl direkt an den RMI Port der Zielmaschine gesandt. Der RMI Empfangsteil des RMI Transportlayers ist in der Lage, den `POST` Befehl zu analysieren und die eigentliche RMI Information zu extrahieren.

2. falls der Firewall Proxy HTTP Befehle nur an Wenserver (Port 80) sendet, dann muss mit Hilfe eines CGI Skriprs auf der Zielmaschine der `POST` Inhalt an das RMI System weitergeleitet werden.

1.4.5.2. Default SocketFactory

Der RMI Transport ist eine Erweiterung der `java.rmi.server.RMISocketFactory` Klasse. Diese stellt eine default Implementation einer Socket-Factory zur Verfügung, welche auch das Firewall Tunnelling zur Verfügung stellt:

- Client Sockets versuchen automatisch eine HTTP Verbindung aufzubauen, sofern der Host nicht direkt mit einer Socketverbindung erreich werden kann.
  Für die Client Sockets ist `java.rmi.server.RMISocketFactory.createSocket` Methode zuständig.

- Server Sockets erkennen automatisch HTTP basierte Verbindungen und extrahieren den `POST` Body für die Weiterbearbeitung innerhalb RMI. Zudem werden in diesem Falle Antworten vom RMI System automatisch als HTTP `POST` Antworten verschlüsselt.
  Für die Server Sockets ist `java.rmi.server.RMISocketFactory.createServerSocket` Methode zuständig.

1.4.5.3. Konfiguration des Client

Es werden keine spezielle Konfigurationen für das Senden von RMI Meldungen mit Hilfe von HTTP durch eine Firewall benötigt.

Der Client kann allerdings diese Art der Kommunikation aktiv verbieten, durch Setzen von `java.rmi.server.disableHttp` auf `true`.

1.4.5.4. Konfiguration des Servers

Damit die Kommunikation durch eine Firewall problemlos funktioniert, sollte der Host **nicht** mit Hilfe einer IP Adresse angegeben werden, da Firewall Proxys den Host dann eventuell nicht finden können, oder die Kommunikation nicht aufbauen.

1. damit der Client eine Methode des Servers aufrufen kann, muss der Client in der Lage sein den Server zu finden. Damit dies möglich ist, wird beim exportieren von remote Referenzen (vom Server) ein voll Qualifizierter Namen des Servers mitgeliefert. Je nach Server Platform und dem Netzwerkumfeld steht diese Information der VM auf dem Server zur Verfügung. Falls nicht, dann muss der Hostnamen als Eigenschaft (Property) beim Serverstart angegeben werden : Property `java.rmi.server.hostname.`

   Beispiel:
   ```
   java -Djava.rmi.server.hostname=hackpack.joller.ch ServerImpl
   ```

2. falls der Server die Weiterleitung von RMI Aufrufen durch die Firewall an einen beliebigen Port nicht unterstützt, kann man folgende Konfiguration benutzen:

   a) der HTTP hört auf Port 80
   b) mit Hilfe des CGI Skripts java-rmi.cgi, welches mit RMI ausgeliefert wird, werden die HTTP POST Meldungen umgeleitet:
   - das Skript führt eine RMI Transportlayer Methode aus, die alle Anfragen an das RMI System weiterleitet.
   - definiert Eigenschaften in der JVM analog zu den Werten und Namen im CGI Encironment

   An Stelle des CGI Skripts kann man natürlich auch ein Servlet einsetzen:
   Sun stellt einen RMIServletHandler zur Verfügung
   http://java.sun.com/products/jdk/1.2/docs/guide/rmi/archives/rmiservlethandler.zip

1.4.5.5. Performance Fragen und Grenzen

Alle HTTP basierten Verbindungen sind mindestens einen Faktor langsamer als jene, welche RMI direkt benutzen können. RMI Verbindungen sind, nachdem sie einmal aufgesetzt wurden, recht schnell, da sie auf direkten Socketverbindungen basieren.

1.4.6. Anhang - java.rmi.cgi

```
#!/bin/sh
#
# java-rmi.cgi
#
# This file handles rmi requests to download RMI code
# The work is done by the class:
#      sun.rmi.transport.proxy.CGIHandler
# This class supports a QUERY_STRING of the form
# "forward=<port>" with a REQUEST_METHOD of "POST"
# The body of the request will be forwarded to the server (
# as aPOST request) to the port given in the URL. The response
# will be returned to the orginal requester.

# Set the path to include the location of the jdk to run
PATH=/opt/java/bin:$PATH

java   \
   -DAUTH_TYPE=$AUTH_TYPE \
   -DCONTENT_LENGTH=$CONTENT_LENGTH \
   -DCONTENT_TYPE=$CONTENT_TYPE \
   -DDOCUMENT_ROOT=$DOCUMENT_ROOT \
   -DGATEWAY_INTERFACE=$GATEWAY_INTERFACE \
   -DHTTP_ACCEPT="$HTTP_ACCEPT" \
   -DHTTP_CONNECTION=$HTTP_CONNECTION \
   -DHTTP_HOST=$HTTP_HOST \
   -DHTTP_USER_AGENT="$HTTP_USER_AGENT" \
   -DPATH_INFO=$PATH_INFO \
   -DPATH_TRANSLATED=$PATH_TRANSLATED \
   -DQUERY_STRING=$QUERY_STRING \
   -DREMOTE_ADDR=$REMOTE_ADDR \
   -DREMOTE_HOST=$REMOTE_HOST \
   -DREMOTE_IDENT=$REMOTE_IDENT \
   -DREMOTE_USER=$REMOTE_USER \
   -DREQUEST_METHOD=$REQUEST_METHOD \
   -DSCRIPT_NAME=$SCRIPT_NAME \
   -DSERVER_NAME=$SERVER_NAME \
   -DSERVER_PORT=$SERVER_PORT \
   -DSERVER_PROTOCOL=$SERVER_PROTOCOL \
   -DSERVER_SOFTWARE=$SERVER_SOFTWARE \
   sun.rmi.transport.proxy.CGIHandler
```

1.5. RMI - Client Interfaces

Damit Applets oder Applikationen entfernte Objekte verwenden können, müssen die RMI Interfaces des `java.rmi` und der dazugehörenden Packages bekannt sein.

1.5.1. Das Remote Interface

```
package java.rmi;
public interface Remote {}
```

Das `java.rmi.Remote` Interface dient dazu, remote Interfaces bekannt zu machen: alle remote Objekte müssen direkt oder indirekt dieses Interface implementieren.

Implementationsklassen können eine beliebige Anzahl remote Interfaces implementieren und andere remote Implementationsklassen erweitern. RMI stellt einige nützliche Klassen zur Verfügung, mit deren Hilfe entfernte Objekte effizient genutzt werden können. Diese Klassen sind :

- `java.rmi.server.UnicastRemoteObject` und
- `java.rmi.activation.Activatable`.

1.5.2. Die RemoteException Klasse

Die Klasse `java.rmi.RemoteException` ist eine Superklasse / Oberklasse für viele der Kommunikations-orientierten Ausnahmen, die beim Ausführen eines entfernten Methodenaufrufes eintreten können. Jede Methode eines remote Interfaces muss die `RemoteException` oder eine ihrer Oberklassen, wie zum Beispiel `java.io.IOException` oder `java.lang.Exception` abfangen.

```
package java.rmi;
public class RemoteException extends java.io.IOException
{
      public Throwable detail;
      public RemoteException();
      public RemoteException(String s);
      public RemoteException(String s, Throwable ex);
      public String getMessage();
      public void printStackTrace();
      public void printStackTrace(java.io.PrintStream ps);
      public void printStackTrace(java.io.PrintWriter pw);
}
```

Eine `RemoteException` besitzt mehrere Konstruktoren :

- einen Konstruktor mit einer definierten Meldung *s*
- einen Konstruktor mit einer definierten Meldung *s* und zusätzlich noch einer verschachtelten Ausnahme *ex*. Typischerweise ist `ex` eine I/O Exception.

Die `getMessage` Methode liefert eine detaillierte Meldung über die verschachtelten Ausnahmen, sofern welche definiert wurden.

Die `printStackTrace` Methode überschreibt die entsprechende Methode der Klasse `java.lang.Throwable` und druckt das Stack Trace der verschachtelten Ausnahme aus.

1.5.3. Die Naming Klasse

Die `java.rmi.Naming` Klasse stellt Methoden zur Verfügung, mit deren Hilfe Referenzen auf entfernte Objekte im Register der entfernten Objekte abgespeichert oder nachgesehen werden können. Die `Naming` Klassenmenthoden verwenden einen URL ähnlichen Namen als Adresse für die entfernten Objekte. Der Namen ist eine **Zeichenkette**, kein URL Objekt, also eine Instanz der `java.lang.String` Klasse mit folgendem Aufbau:

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

wobei
host den Host beschreibt (remote oder lokal) auf dem die Registry, das Register, läuft und
port die Portnummer ist, an der die Registry Calls erwartet.
name ist eine Zeichenkette, welche von der Registry nicht interpretiert wird.

Sowohl *host* als auch *port* sind optional. Falls *host* fehlt, wird als Standardwert **`localhost`** angenommen (da die Registry auf dem Server laufen muss). Falls *port* fehlt, wird der Standardwert **1099** abgenommen, der Standardport von RMI's `rmiregistry`.

Binding das Binden, besteht darin, dass man einen Namen eines entfernten Objekts einem Namen zugeordnet. Unter diesem Namen kann das remote Objekt später, nach dem Binden, im Register nachgeschlagen werden (man kann auch eine Liste aller eingetragenen Namen ausgeben lassen : `Naming.list`). Das Zuordnen eines Namens zu einem entfernten Objekt kann mit Hilfe zweier Methoden geschehen :

- `bind`, für das erstmalige Binden. Falls der Eintrag bereits existiert, wird eine Ausnahme geworfen.
- `rebind`, für das Überschreiben eines Eintrages. Falls der Eintrag noch nicht existiert, funktioniert `rebind()` wie `bind()`, es wird also ein Ersteintrag durchgeführt.

Sobald ein Objekt bei einem RMI Register auf dem Serverhost registriert ist (gebunden), können lokale und entfernte Clients das Register nach registrierten Objekten durchsuchen und deren Methoden einsetzen.

Ein Register kann von mehreren Servern oder auch nur von einem Server benutzt werden. Die Methode `java.rmi.registry.LocateRegistry.createRegistry` beschreibt diesen Aspekt im Detail.

```
package java.rmi;
public final class Naming {
      public static Remote lookup(String url)
      throws NotBoundException, java.net.MalformedURLException,
      RemoteException;
      public static void bind(String url, Remote obj)
      throws AlreadyBoundException,
      java.net.MalformedURLException, RemoteException;
      public static void rebind(String url, Remote obj)
      throws RemoteException, java.net.MalformedURLException;
      public static void unbind(String url)
      throws RemoteException, NotBoundException,
      java.net.MalformedURLException;
      public static String[] list(String url)
      throws RemoteException, java.net.MalformedURLException;
}
```

Die `lookup` Methode liefert das remote Objekt zum Dateinamen. Falls der Namen nicht an ein Objekt gebunden ist, wird die `NotBoundException` geworfen.

Die `bind` Methode bindet den spezifizierten Namen an ein entferntes Objekt. Falls der Namen bereits registriert ist, wird eine `AlreadyBoundException` geworfen.
.
Die `rebind` Methode bindet das Objekt an den Namen, auch wenn der Namen bereits registriert wurde. Der alte Namen wird einfach überschrieben.

Die `unbind` Methode entfernt die Bindung eines Namens an ein entferntes Objekts.Die Methode wirft die `NotBoundException` falls die Bindung nicht gefunden wird.
.
Die `list` Methode liefert ein Array von Zeichenketten, der Objekte, die registriert sind., in URL Form. Nur Host und Port der URL werden benötigt. Der Dateiteil der URL wird ignoriert.

Bemerkung – Die `java.rmi.AccessException` kann auch in vielen andern Fällen geworfen werden, als Ergebnis irgend eines Methodenaufrufs. Dies `AccessException` zeigt, dass der Aufrufer keine Erlaubnis hat,die spezifizierte Operation auszuführen.

Es könnte beispielsweise sein, dass der Client nur Zugriff auf die Methoden hat, sofern er lokal ist. `bind, rebind` und `unbind` und `lookup` Operationen werden typischerweise von entfernten Clients aufgerufen

1.5.4. Aufgabe

Bauen Sie ein bestehendes RMI Programm so um, dass es vor und nach dem Binden des remote Objekts jeweils eine Liste der registrierten Objekte ausgibt.

1.6. RMI - Server Interfaces

1.6.1. Die RemoteObject Klasse

Die `java.rmi.server.RemoteObject` Klasse implementiert das Verhalten von `java.lang.Object` für remote Objekte. Die Methoden `hashCode` und `equals` stehen auch für entfernte Objekte zur Verfügung und erlauben die Speicherung in Hash Tabellen und den Objektvergleich. Die `equals` Method liefert den Wert `true` falls zwei Remote Objekte sich auf das selbe entfernte Objekt beziehen. Der Vergleich bezieht sich auf die Referenzen auf entfernte Objekte. Die `toString` Method liefert eine Zeichenkette, welche das entfernte Objekt beschreibt. Inhalt und Syntax dieser Zeichenkette sind je nach Implementation leicht unterschiedlich, entsprechend der Syntax des Gastbetriebsystems. Alle andern Methods von `java.lang.Object` bleiben unverändert.

```
package java.rmi.server;
public abstract class RemoteObject
implements java.rmi.Remote, java.io.Serializable
{
      protected transient RemoteRef ref;
      protected RemoteObject();
      protected RemoteObject(RemoteRef ref);
      public RemoteRef getRef();
      public static Remote toStub(java.rmi.Remote obj)
      throws java.rmi.NoSuchObjectException;
      public int hashCode();
      public boolean equals(Object obj);
      public String toString();
}
```

Da die `RemoteObject` Klasse abstrakt ist, kann sie nicht instanziert werden. Die Konstruktoren der Klasse müssen also von einer Unterklasse, welche die abstrakte Klasse erweitert und implementiert, aufgerufen werden.

Der erste `RemoteObject` Konstruktor kreiert ein `RemoteObject` mit einer leeren remote Referenz. Der zweite `RemoteObject` Konstruktor kreiert ein `RemoteObject` bei gegebener remote Referenz, *ref*.

Die `getRef` Methode liefert die remote Referenz für das remote Objekt.

Die `toStub` Method liefert einen Stub für ein entferntes Objekt *obj*, welches als Parameter übergeben wurde. Diese Methode ist nur einsetzbar, wenn die remote Objekt Implementation bereits exportiert wurde. Falls der Stub für das entfernte Objekt nicht gefunden werden kann, wird die `NoSuchObjectException` geworfen.

1.6.1.1. Objekt Methoden, welche von der RemoteObject Klasse Überschrieben werden

Die Defaultimplementatonen der `java.lang.Object` Klasse für die `equals`, `hashCode` und `toString` Methoden müssen den gegebenheiten der entfernten Objekte angepasst werden. Daher stellt die `RemoteObject` Klasse Implementationen für diese Methoden zur Verfügung, die besser geeignet sind für remote Objekte.

1.6.1.1.1. equals und hashCode Methoden

Damit ein entferntes Objekt als Schlüssel für eine Hashtabelle verwendet werden kann, müssen die Methoden `equals` und `hashCode` für den Einsatz in einem verteilten Umfeld überschrieben werden. Dies geschieht in der Klasse `java.rmi.server.RemoteObject`:

- Die `java.rmi.server.RemoteObject` Klasse implementiert die *`equals`* Method, mit deren Hilfe festgestellt werden kann, ob zwei Objektreferenzen gleich oder ungleich sind, also das gleiche oder zwei unterschiedliche Objekte beschreiben.
- Die `java.rmi.server.RemoteObject` Klasse implementiert die *`hashCode`* Methode, welche den selben Wert zurück liefern, sofern die entfernten Objekte identisch sind.

1.6.1.1.2. toString Methode

Die `toString` Methode ist so definiert, dass sie eine Zeichenkette zurück liefert. Diese stellt eine remote Reference für das Objekt dar. Je nach Objektreferenztypus hat die Zeichenkette einen spezifischen Aufbau. Die aktuelle Implementation für ein Singelton (Unicast) Objekt umfasst eine Objektidentifikation und zusätzliche Informationen zum Transportlayer (Hostnamen, Portnummer).

1.6.1.1.3. clone Methode

Objekte sind nur clonebar, falls sie die `java.lang.Cloneable` Schnittstelle unterstützen. Stubs für remote Objekte werden mit Hilfe des `rmic` Compilers deklariert und sind als final deklariert. Sie implementieren das `Cloneable` Interface nicht. Somit ist ein Clonen eines Stubs nicht möglich.

1.6.1.2. Serialisierte Form

Die `RemoteObject` Klasse implementiert die speziellen, privaten `writeObject` und `readObject` Methoden, welche von der Objektserialisierung verwendet werden, um Daten (Objekte) mit dem `java.io.ObjectOutputStream` zu verknüpfen. `RemoteObject`'s serialisierte Form wird mit Hilfe der Methode:

```
private void writeObject(java.io.ObjectOutputStream out)
throws java.io.IOException, java.lang.ClassNotFoundException;
```

geschrieben:

- falls die Referenz auf das entfernte Objekt `RemoteObject, ref, null` ist, wirft die Methode die Ausnahme `java.rmi.MarshalException`.
- falls die remote Referenz, *ref*, nicht null ist:
  - *ref* 's Klasse wird mit Hilfe der Methode `getRefClass` bestimmt. Typischerweise liefert diese Methode einen Namen der remote Referenz.

    Falls der zurück gelieferte Namen nicht null ist
    - *ref* 's Klassen Name wird in den Stream, *out*,in UTF Format, geschrieben.
    - *ref* 's `writeExternal` Methode wird aufgerufen, mit *out* als Parameter,so dass *ref* die externe Darstellung des Objektes in den Strom schreiben kann.

- falls die `ref.getRefClass` null zurück liefert:
  - wird eine leere Zeichenkette in UTF Format in den Strom *out* geschrieben.
  - *ref* wird in serialisierter Form mit Hilfe der Methode writeObject in den Strom *out* geschrieben.

Ein `RemoteObject` wird mit Hilfe der folgenden Methode der Klasse `ObjectInputStream` aus dem Strom gelesen und rekonstruiert (im Laufe der Deserialisierung):

```
private void readObject(java.io.ObjectInputStream in)
throws java.io.IOException, java.lang.ClassNotFoundException;
```

- Zuerst wird der Klassennamen von *ref* als UTF Zeichenkette gelesen.

  Falls der Klassennamen eine leere Zeichenkette ist:
  - ein Objekt wird aus dem Strom gelesen und *ref* wird entsprechend initialisiert, mit Hilfe der `in.readObject` Methode.

  Falls die Klasse eine nichtleere Zeichenkette ist:
  - *ref* 's vollständiger Klassennamen wird rekonstruiert, als Konkatenation der Werte von `java.rmi.server.RemoteRef.packagePrefix` und "." mit dem Klassennamen, der aus dem Strom gelesen wurde
  - eine Instanz der ref's Klasse wird kreiert
  - die neue Instanz liest ihre externe Darstellung aus dem *in* Strom.

1.6.2. Die RemoteServer Klasse

Die `java.rmi.server.RemoteServer` Klasse ist die Oberklasse für alle Server Implementationsklassen, `java.rmi.server.UnicastRemoteObject` und `java.rmi.activation.Activatable`.

```
package java.rmi.server;
public abstract class RemoteServer extends RemoteObject {
      protected RemoteServer();
      protected RemoteServer(RemoteRef ref);
      public static String getClientHost()
            throws ServerNotActiveException;
      public static void setLog(java.io.OutputStream out);
      public static java.io.PrintStream getLog();
}
```

Da die `RemoteServer` Klasse abstrakt ist, kann sie nicht instanziert werden. Daher muss einer der Konstruktoren der Klasse RemoteServer aus einer implementierenden Unterklasse aufgerufen werden. Der erste Konstruktor von `RemoteServer` kreiert einen `RemoteServer` mit einer null remote Referenz. Der zweite `RemoteServer` Konstruktor kreiert einen `RemoteServer` mit einer gegebenen remote Referenz, *ref*.

Die `getClientHost` Methode erlaubt einer aktiven Methode die Bestimmung des Hosts, der die remote Methode im aktiven Thread initialisierte. Falls im aktiven Thread keine remote Methode aktiv ist, wird die Ausnahme `ServerNotActiveException` geworfen

Die `setLog` Methode logged RMI Aufrufe in einen speziellen Ausgabestrom.Falls die Angabe zum Ausgabestrom fehlt, wird das Logging ausgeschaltet.

Die `getLog` Methode liefert den Strom der RMI Logs..

1.6.3. Die UnicastRemoteObject Klasse

Die Klasse `java.rmi.server.UnicastRemoteObject` unterstützt das Kreieren und Exportieren entfernter Objekte. Die Klasse implementiert einen entfernten Server mit folgenden Charakteristiken:

- Referenzen auf solche Objekte sind maximal so lange gültig, wie der Prozess, der das entfernte Objekt kreiert hat, am Leben ist.
- Die Kommunikation geschieht mit mit Hilfe eines TCP Transports.
- Aufrufparameter und Ergebnisse nutzen ein Stromprotokoll, um zwischen Client und Server zu kommunizieren.

```
package java.rmi.server;
public class UnicastRemoteObject extends RemoteServer {
      protected UnicastRemoteObject()
                  throws java.rmi.RemoteException;
      protected UnicastRemoteObject(int port)
                  throws java.rmi.RemoteException;
      protected UnicastRemoteObject(int port,
            RMIClientSocketFactory csf,
            RMIServerSocketFactory ssf)
                  throws java.rmi.RemoteException;
      public Object clone()
                  throws java.lang.CloneNotSupportedException;
      public static RemoteStub exportObject(java.rmi.Remote obj)
                  throws java.rmi.RemoteException;
      public static Remote exportObject(java.rmi.Remote obj,
            int port)
                  throws java.rmi.RemoteException;
      public static Remote exportObject(Remote obj, int port,
            RMIClientSocketFactory csf,
            RMIServerSocketFactory ssf)
                  throws java.rmi.RemoteException;
      public static boolean unexportObject(java.rmi.Remote obj,
            boolean force)
                  throws java.rmi.NoSuchObjectException;
}
```

1.6.3.1. Konstruktiion eines neuen Remote Objekts

Eine remote Objekt Implementation (also eine Implementation von einem oder mehreren remote Interfaces) muss kreiert und exportiert werden. Exportieren eines remote Objekts sorgt dafür, dass dieses Objekt für ankommende Anrufe bereit ist. Konkret besagt dies, dass ein remote Objekt, welches als ein `UnicastRemoteObject` exportiert wird, an einem TCP Port auf ankommende Anfragen hört, wobei an einem Port mehr als ein Objekt "hören" darf. Eine Implementation eines remote Objekts kann die Klasse `UnicastRemoteObject` erweitern und deren Konstruktor benutzen, um das Objekt zu exportieren, oder aber es kann auch eine andere Klasse erweitern, und das Objekt mit Hilfe der Methoden `exportObject` der Klasse `UnicastRemoteObject` exportieren.

Der argumentlose (=Default) Konstruktor kreiert und exportiert zur Laufzeit ein remote Objekt an einen anonymen (oder beliebig gewählten) Port.

Die zweite Form des Konstruktors besitzt ein einziges Argument, *port*, gibt den Port an, an dem eintreffende Anrufe empfangen werden.

Der dritte Konstruktor empfängt Anfragen an einem spezifizierten Port *port* via einen `ServerSocket,` der von der `RMIServerSocketFactory` kreiert wurde; Clients werden Verbindungen zum remote Objekt mit Hilfe des Sockets aufbauen, der von der `RMIClientSocketFactory` geliefert wird..

1.6.3.2. Export einer Implementation, die nicht RemoteObject erweitert

Die `exportObject` Methode (in irgend einer Form) wird zum Exportieren von einfachen peer-to-peer remote Objekten, welche nicht als Erweiterungen der `UnicastRemoteObject` Klasse definiert wurden.

Die erste Form der `exportObject` Methode besitzt als Parameter *obj*, das remote Objekt, welches eintreffende RMI Calls entgegen nimmt; diese `exportObject` Methode exportiert das Object an einem anonymen (oder beliebigen) Port, der zur Laufzeit festgelegt wird.

Die zweite Form der `exportObject` Methode verwendet zwei Parameter, ein remote Objekt, *obj*, und *port*, den Port, an dem das remote Objekt auf eintreffende Anfragen wartet.

Die dritte Form der `exportObject` Methode exportiert das Objekt, *obj*, mit der spezifizierten `RMIClientSocketFactory`, *csf*, und `RMIServerSocketFactory`, *ssf*,an den spezifizierten Port *port*. Das Objekt *muss* exportiert sein, bevor es zum ersten Mal in einem RMI Call entweder als Parameter oder als Rückgabewert eingesetzt wird. Sonst wird eine `java.rmi.server.StubNotFoundException` Ausnahme geworfen, falls ein remote Call versucht dieses "nicht exportierte" remote Objekt als Argument oder Rückgabewert einzusetzen.

Einmal exportiert, kann ein Objekt als Argument eines RMI Calls oder als Ergebnis eines RMI Calls eingesetzt werden. Die `exportObject` Methode liefert einen `Remote` Stub, das Stub Objekt für das remote Objekt, `obj`, welcher an Stelle des remote Objekts in einem RMI Call verwendet wird.

1.6.3.3. Verwendung eines UnicastRemoteObject in einem RMI Call

Wie bereits erwähnt, wird falls ein Objekt vom Typ `UnicastRemoteObject` als Parameter oder Rückgabewert in einem RMI Call verwendet wird, das Objekt durch den Stub des entfernten Objekts ersetzt. Die Implementation eines remote Objekts bleibt in der virtuellen Maschine, in der es kreiert wurde. Es bewegt sich also nicht. Mit andern Worten: ein remote Objekt wird als Referenz (*by reference*) nicht als Wert (*by value*) übergeben.

1.6.3.4. Serialisierung eines UnicastRemoteObject

Die Informationen, die in einem `UnicastRemoteObject` enthalten ist, sind *transient* (=*temporär*; im Gegensatz zu permanent, *beständig*, *persistent*). Daher wird die Information auch nicht gespeichert, falls ein solches Objekt in einen benutzerdefinierten `ObjectOutputStream` geschrieben wird (zum Beispiel, falls das Objekt in eine Datei geschrieben wird, mit Hilfe der Serialisierung).

Ein Objekt, welches eine Instanz einer benutzerdefinierten Unterklasse von `UnicastRemoteObject` ist, kann nicht-transiente (also persistente, beständige) Daten enthalten, welche gespeichert werden können, falls das Objekt serialisiert wird.

Falls ein `UnicastRemoteObject` aus einem `ObjectInputStream` gelesen wird, wird es automatisch an das RMI Laufzeitsystem exportiert und ist damit in der Lage RMI Calls zu empfangen. Falls das Exportieren des Objekts aus irgend einem Grunde misglückt, endet die Deserialisierung mit einer Ausnahme.

1.6.3.5. Unexporting ein UnicastRemoteObject

Die `unexportObject` Methode sorgt dafür, dass das remote Objekt, *obj*, für eintreffende Anfragen nicht mehr verfügbar ist.

Falls der Parameter `force true` ist, wird das Objekt "zwangs unexportiert", selbst falls noch pendente Anfragen an entfernte Objekte bestehen oder in Bearbeitung sind.

Falls der `force` Parameter `false` ist, wird das Objekt nur dann unexportiert (=für entfernte Anfragen nicht mehr verfügbar), falls keine Anfragen pendent oder in Bearbeitung sind.

Falls das Objekt erfolgreich unexportiert wurde, entfernt das RMI Laufzeitsystem das Objekt aus seiner internen Tabelle. Diese Art und Weise, Objekte remote nicht mehr zur Verfügung zu stellen (zu "unexportieren") kann dazu führen, dass einzelne Clients in einem Wartezustand verbleiben. Diese Methode wirft die `java.rmi.NoSuchObjectException` Ausnahme, falls das Objekt zuvor nicht exportiert, also dem RMI Laufzeit zur Verfügung gestellt wurde.

1.6.3.6. Die clone Methode

Objekte sind nur mit Hilfe der Standardmechanismen der Java Sprache clonebar, falls sie das Interface `java.lang.Cloneable` unterstützen. Die Klasse `java.rmi.server.UnicastRemoteObject` unterstützt diese Schnittstelle nicht, implementiert aber eine `clone` Methode. Somit können Unterklassen, falls nötig, die Schnittstelle `Cloneable` implementieren, um remote Objekte clonebar zu machen. Die `clone` Methode kann von Subklassen benutzt werden, um geklonte remote Objekte zu kreieren. Diese besitzen initial den selben Inhalt wie das Original. Das geklonte Objekt wird aber exportiert und somit für entfernte Anfragen verfügbar. Damit verändert es seinen Zustand und wird sich somit in der Regel vom ursprünglichen Objekt unterscheiden.

1.6.4. Das Unreferenced Interface

```
package java.rmi.server;
public interface Unreferenced {
      public void unreferenced();
}
```

Das `java.rmi.server.Unreferenced` Interface ermöglicht es einem Server Objekt eine Nachricht zu erhalten, falls keine Clients mehr remote Referenzen auf den Server besitzen. Der Distributed Garbage Collection Mechanismus unterhält eine Liste (die Referenzmenge) für jedes remote Objekt, mit allen Client Virtual Machinen, welche Referenzen auf dieses remote Objekt besitzen. So lange Clients eine remote Referenz auf ein remote Objekt besitzen, unterhält das RMI Laufzeitsystem eine lokale Referenz zu diesem remote Objekt. Falls nun diese "reference" Menge leer wird, wird die `Unreferenced.unreferenced` Method aufgerufen, sofern der Server das `Unreferenced` Interface implementiert. Es ist *kein* remote Objekt nötig, um das `Unreferenced` Interface zu implementieren.

Solange lokale Referenzen auf ein remote Objekt existieren, können diese auch in Methodenaufrufen und als Rückgabewerte verwendet werden. Jeder Prozess, welcher eine Referenz erhält, wird in die Referenzmenge für das remote Objekt eingetragen. Sobald die Referenzmenhe leer wird, wird die `unreferenced` Methode des remote Objektes aufgerufen. Die `Unreferenced` Method kann auch mehr als einmal aufgerufen werden, immer wenn die Referenzmenge leer wird. Remote Objekte werden nur dann geöscht, falls keine Referenzen mehr existieren (lokale oder von Clients gehaltene).

1.6.5. Die RMISecurityManager Klasse

```
package java.rmi;
public class RMISecurityManager extends
          java.lang.SecurityManager {
     public RMISecurityManager();
     public synchronized void checkPackageAccess(String pkg)
          throws RMISecurityException;
}
```

Der `RMISecurityManager` stellt die selben Sicherheitsmechanismen zur Verfügung wie der `java.lang.SecurityManager`, wobei die `checkPackageAcess` Methode überschrieben wird. In RMI Applikationen können Stubs und Klassen nur vom lokalen Classpath geladen werden, falls kein Security Manager geladen wurde. Dies stellt sicher, dass die Applikationen vor (mit RMI) herunter geladenem Code geschützt ist.

1.6.6. Die RMIClassLoader Klasse

Die java.rmi.server.RMIClassLoader Klasse stellt public static Methoden für Netzwerk-basiertes Laden von Klassen in RMI zur Verfügung. Diese Methoden werden von RMI's internen Marshal Streams aufgerufen, um das dynamische Laden von Klassen (RMI Parameter und Rückgabewerte) zu ermöglichen. Man kann diese Methoden aber auch direkt aus Applikationen aufrufen, um das Klassenladen von RMI nachzuahmen.

Die RMIClassLoader Klass besitzt keine öffentlich zugägnlichen Konstruktoren und kann somit nicht instanziert werden.

```
package java.rmi.server;
public class RMIClassLoader {
     public static String getClassAnnotation(Class cl);
     public static Object getSecurityContext(ClassLoader
                                                  loader);
     public static Class loadClass(String name)
          throws java.net.MalformedURLException,
          ClassNotFoundException;
     public static Class loadClass(String codebase, String
                                                    name)
          throws java.net.MalformedURLException,
          ClassNotFoundException;
     public static Class loadClass(URL codebase, String name)
          throws java.net.MalformedURLException,
          ClassNotFoundException;
}
```

Die `getClassAnnotation` Methode liefert eine `String` Darstellung des Netzwerk Codebase Pfades, den ein remote Endpoint benutzen sollte, um die Definition der angegebenen Klasse herunter zu laden. Das RMI Laufzeitsystem benutzt `String` Objekte dieser Methode, als Zuätze für die Klassenbeschreibung in den marshal Streams. Das Format dieser Codebase Zeichenkette ist ein Pfad (URL Zeichenketten, mit Hilfe von Leerzeichen getrennt).

Der Aufbau der Codebase Zeichenkette hängt vom ClassLoader der Klasse ab:

- falls der Class Loader einer der folgenden ist:
  - der "system class loader" (der Class Loader, mit dem Klassen im CLASSPATH der Applikation geladen werden, und der von der Methode `ClassLoader.getSystemClassLoade`r zurück gegeben wird),
  - eine Oberklasse des "system class loader",
  - oder null (der "boot class loader" mit dem VM Klassen geladen werden),

  dann wird der Wert der `java.rmi.server.codebase` Eigenschaft zurück gegeben, oder null, falls diese Eigenschaft nicht gesetzt wurde.
- sonst, falls der Class Loader eine Instanz der Klasse `java.net.URLClassLoade`r ist, dann besteht die Codebase Zeichenkette aus einer durch Leerzeichen getrennten Liste von URLs, die von den `getURLs` Methoden von den Class Loadern zurück gegeben werden. Falls der `URLClassLoader` vom RMI Laufzeitsystem kreiert wurde, um eine der `RMIClassLoader.loadClass` Methoden einzusetzen, dann werden keine

Privilegien benötigt, um die Codebase Zeichenkette zu erhalten. Falls es sich um eine beliebige `URLClassLoader` Instanz handelt, muss der Aufrufer der Methode: `openConnection().getPermission()`.

- falls der Class Loader keine Instanz des `URLClassLoade`r ist, dann wird der Wert der `java.rmi.server.codebase` Eigenschaft zurück gegeben, oder `null` falls die Eigenschaft nicht gesetzt ist..

Die `getSecurityContext` Methode ist verworfen worden, weil das Security Modell in JDK 1.2 geändert wurde. Falls der Class Loader vom RMI Laufzeitsystem kreiert wurde, um eine der `RMIClassLoader.loadClass` Methoden auszuführen, dann wird die erste URL im Class Loader Codebase Pfad zurück gegeben; sonst wird null zurück gegeben.

Die drei `loadClass` Methoden versuchen alle, die Klasse mit dem spezifizierten Namen mit Hilfe des Class Loaders des aktuellen Threads zu laden. Falls ein Security Manager gesetzt wurde, wird ein interner `URLClassLoader` für einen bestimmten Codebase Pfad eingesetzt (abhängig von der Methode):

- Die `loadClass` Methode mit einem Parameter (dem Klassennamen *nam*e) verwendet implizit den Wert der Eigenschaft `java.rmi.server.codebase,` mit der der Codebase Pfad, welcher eingesetzt werden soll, spezifiziert wird. Diese Version der `loadClass` Methode wurde verworfen, weil das Setzen der Eigenschaft (auf der Kommandozeile) `java.rmi.server.codebase` durch eine bessere Konstruktion ersetzt wurde:
- Die `loadClass` Methode mit der Zeichenkette `String` *codebase* als Parameter verwendet diesen Parameter als Codebase Pfad; diese Zeichenkette muss eine durch Leerzeichen getrennte Liste von URLs sein. Diese Liste wird von der Methode `getClassAnnotation` angezeigt.
- Die `loadClass` Methode mit dem `java.net.URL` *codebase* Parameter verwendet diese `URL` als Codebase.

Für alle `loadClass` Methoden wird der Codebase Pfad zusammen mit dem Class Loader des aktuellen Threads eingesetzt, um die interne Class Loader Instanz zu bestimmen, mit der versucht wird, die Klasse zu laden. Das RMI Laufzeitsystem unterhält eine Tabelle interner Class Loader Instanzen. Index dieser Tabelle sind Paare <Class Loader Oberklasse; Loaders Codebase Pfad [URL]> .

Eine `loadClass` Methode schaut in dieser Tabelle nach, welcher `URLClassLoader` zum gewünschten Index gehört (<Class Loader OberklasseCodebasis>). Falls kein solcher Class Loader existiert, wird einer kreiert und in die Tabelle eingeführt..

Die `loadClass` Methode mit einem spezifizierten Namen *nam*e wählt erwartungsgemäss den Class Loader zum spezifizierten Namen.

Sofern ein Security Manager angegeben wurde, also ein Aufruf der Methode `System.getSecurityManager` nicht null zurück gibt, dann muss der Aufrufer der Methode bestimmte Privilegien besitzen: er muss mindestens das "connect" Recht besitzen.Sonst wird eine `ClassNotFoundException` Ausnahme geowrfen werden.

Falls kein Security Manager installiert wurde, dann wird unter JDK1.2+ der Codebase Pfad ignoriert und versucht die Klasse mit Hilfe des Namens zu finden und mit Hilfe des Class Loaders des aktuellenThreads zu laden.

1.6.7. Das LoaderHandler Interface

```
package java.rmi.server;
public interface LoaderHandler {
     Class loadClass(String name)
          throws MalformedURLException, ClassNotFoundException;
     Class loadClass(URL codebase,String name)
          throws MalformedURLException, ClassNotFoundException;
     Object getSecurityContext(ClassLoader loader);
}
```

Bemerkung – Das `LoaderHandler` Interface wurde inJDK 1.2 verworfen, da es in JDK1.1 lediglich für RMI Implementationen verwendet wurde.

1.6.8. RMI Socket Factories

Falls das RMI Laufzeitsystem eine Instanz der Klassen `java.net.Socket` oder `java.net.ServerSocket` zur Kommunikation benötigt, werden an Stelle der direkten Konstruktoren (für Client Sockets und Server Sockets) `createSocket` und `createServerSocket` Methoden des `RMISocketFactory` Objekts verwendet. Die aktuelle `RMISocketFactory` lässt sich mit Hilfe der (statischen) Methode `RMISocketFactory.getSocketFactory` bestimmt.

Damit hat der Programmierer die Möglichkeit eigene Socket Factories einzubauen, für spezielle Anwendungen, beispielsweise spezielle Unterklassen von `java.net.Socket` und
`java.net.ServerSocket.`

In JDK 1.1 waren die Möglichkeiten Instanzen der `RMISocketFactory` einzusetzen, ziemlich limitiert.

In JDK 1.2, wurde ein neue Interfaces `RMIServerSocketFactory` und `RMIClientSocketFactory` eingeführt, um flexiblere Kommunikation mit entfernten Objekten aufbauen zu können.

Damit RMI Applikationen die Vorteile dieser Interfaces nutzen können, wurden diverse neue Konstruktoren und `exportObject` Methoden definiert, um die Möglichkeiten der Client und Server Socket Factory und der Klassen `UnicastRemoteObject` und `java.rmi.activation.Activatable` besser nutzen zu können.

Entsprechend werden Objekte, die mit einem der neuen Konstruktoren kreiert wurden anders behandelt. Das RMI Laufzeitsystem weiss, dass in diesem Falle die angepasste `RMIServerSocketFactory` eingesetzt werden muss, um einen `ServerSocket` für eintreffende Anfragen zu kreieren. Auch der gesamte Kommunikationsablauf auf RMI Ebene wird entsprechend angepasst.

1.6.8.1. Die RMISocketFactory Klasse

Die abstrakte `java.rmi.server.RMISocketFactory` Klasse stellt ein Interface zur Verfügung, zur Spezifikation wie Sockets für den Transport beschafft werden sollten. Die unten beschriebene Klasse verwendet die `Socket` und `ServerSocket` vom `java.net` Pakage.

```
package java.rmi.server;
public abstract class RMISocketFactory
     implements RMIClientSocketFactory, RMIServerSocketFactory{
     public abstract Socket createSocket(String host, int port)
          throws IOException;
     public abstract ServerSocket createServerSocket(int port)
          throws IOException;
     public static void setSocketFactory(RMISocketFactory fac)
          throws IOException;
     public static RMISocketFactory getSocketFactory();
     public static void setFailureHandler(RMIFailureHandler
     fh);
     public static RMIFailureHandler getFailureHandler();
}
```

Die statische Methode `setSocketFactory` wird benutzt, um die Socket Factory zu spezifizieren, von der RMI Sockets erhält. Eine Applikation kann diese Methode einmal verwenden, um eine eigene `RMISocketFactory` Instanz zu spezifizieren.

Mögliche Gründe für den Einsatz einer eigenen `RMISocketFactory` könnten sein:

- filtern von Verbindungen und werfen von Ausnahmen im Ausnahmefall
- Einbau eigener Erweiterungen der `java.net.Socket` oder `java.net.ServerSocket` Klassen, zum Beispiel mit zusätzlich geschützter Kommunikation

Eine eigene `RMISocketFactory` kann man nur setzen, falls der Security Manager dies erlaubt.
Ist dies nicht der Fall, wird eine `SecurityException` geworfen.

Die statische Methode `getSocketFactory` liefert die Socket Factory, mit der RMI arbeitet. Falls die Methode null zurück gibt, wurde die Socket Factory nicht gesetzt.

Der RMI Transport Layer verwendet die `createSocket` und `createServerSocket` Methoden der `RMISocketFactory`, jener, die von der Methode `getSocketFactory` zurück gegeben wird.

Zum Beispiel:

```
RMISocketFactory.getSocketFactory().createSocket(m_Host,
m_Port)
```

In diesem Beispiel soll die Methode `createSocket` einen Client Socket liefern, der mit dem Host m_Host und Port m_Port kommuniziert.

Die Methode `createServerSocket` soll einen Server Socket am spezifizierten Port m_Port kreieren.

Die Standard Implementation der `RMISocketFactory` liefert auch die Basis für die RMI Kommunikation durch eine Firewall mit Hilfe von HTTP:

- beim Aufruf von `createSocke`t versucht die Factory eine HTTP Verbindung aufzubauen, falls der Host nicht direkt mit Hilfe der Sockets erreicht werden kann.
- beim Aufruf der Methode `createServerSocke`t liefert die Factory einen Server Socket, der automatisch festellt, ob eine neu akzeptierte Verbindung mit Hilf eines HTTP POST geschehen ist.

Die Methode `setFailureHandler` definiert den Failure Handler, mit dem Fehler beim Kreieren eines Server Sockets durch das RMI Laufzeitsystem abgefangen werden.

Die Methode `getFailureHandler` liefert den aktuellen Handler für Fehler beim Kreieren von Sockets, bzw. null, falls der Failure Handler nicht gesetzt wurde.

1.6.8.2. Das RMIServerSocketFactory Interface

Für die Kommunikation mit entfernten Objekten kann eine `RMIServerSocketFactory` Instanz spezifiziert werden. Dies geschieht beim Exportieren eines remote Objekts entweder mit Hilfe eines entsprechenden `UnicastRemoteObject` Konstruktors oder der `exportObject` Methode oder dem passenden `java.rmi.activation.Activatable` Konstruktor.

Falls einem remote Objekt eine Server Socket Factory zugeordnet ist, wird das RMI Laufzeitsystem diese beim Kreieren eines `ServerSocket` einsetzen (mit Hilfe der `RMIServerSocketFactory.createServerSocket` Methode).

```
package java.rmi.server;
public interface RMIServerSocketFactory {
     public java.net.ServerSocket createServerSocket(int port)
          throws IOException;
}
```

1.6.8.3. Das RMIClientSocketFactory Interface

Für die Kommunikation mit entfernten Objekten kann eine `RMIClientSocketFactory` Instanz spezifiziert werden. Dies geschieht beim Exportieren eines remote Objekts entweder mit Hilfe eines entsprechenden `UnicastRemoteObject` Konstruktors oder der `exportObject` Methode oder dem passenden `java.rmi.activation.Activatable` Konstruktor.

Falls einem remote Objekt eine Server Socket Factory zugeordnet ist, wird das RMI Laufzeitsystem diese beim Kreieren eines `ServerSocket` einsetzen (mit Hilfe der `RMIClientSocketFactory.createSocket` Methode).

```
package java.rmi.server;
public interface RMIClientSocketFactory {
     public java.net.Socket createSocket(String host, int port)
          throws IOException;
}
```

1.6.9. Das RMIFailureHandler Interface

Das `java.rmi.server.RMIFailureHandler` Interface stellt eine Methode zur Verfügung, um anzugeben, wie das RMI Laufzeitsystem reagieren soll, falls beim Kreieren eines Server Sockets ein Fehler auftritt (beim Exportieren des Objektes).

```
package java.rmi.server;
     public interface RMIFailureHandler {
     public boolean failure(Exception ex);
}
```

Die `failure` Methode wird mit der Ausnahme als Parameter aufgerufen, welche das RMI Laufzeitsystem daran gehindert hat, einen `java.net.ServerSocke`t zu kreieren. Die Methode liefert true, falls das Laufzeitsystem einen weieern Versuch starten sollte, falls sonst.

Bevor die Methode aufgerufen werden kann, muss ein Failure Handler mit Hilfe der Methode `RMISocketFactory.setFailureHandler` worden sein. Falls kein Handler spezifiziert wurde, versucht das RMI Laufzeitsystem nach einer kurzen Wartezeit den `ServerSocket` erneut zu konstruieren.

Falls der `ServerSocket` beim Exportieren des Objekts ausfällt, dann wird der `RMIFailureHandler` nicht geworfen.

1.6.10. Die LogStream Klasse

Die Klasse `LogStream` stellt einen Mechanismus zur Verfügung, mit dessen Hilfe Fehler, die von Interesse für einen System Monitor sein könnten, gelogged werden. Intern werden damit die Server Aufrufe gelogged.

```
package java.rmi.server;
public class LogStream extends java.io.PrintStream {
      public static LogStream log(String name);
      public static synchronized PrintStream getDefaultStream();
      public static synchronized void setDefaultStream(
      PrintStream newDefault);
      public synchronized OutputStream getOutputStream();
      public synchronized void setOutputStream(OutputStream out);
      public void write(int b);
      public void write(byte b[], int off, int len);
      public String toString();
      public static int parseLevel(String s);
// constants for logging levels
      public static final int SILENT = 0;
      public static final int BRIEF = 10;
      public static final int VERBOSE = 20;
}
```

Bemerkung – Die `LogStream` Klasse wurde in JDK 1.2+ verworfen.

Die Methode `log` liefert einen LogStream (zumindest bei gegebenem Namen). Falls kein Log diesem Namen entspricht, wird ein Log kreiert.

Die Methode `getDefaultStream` liefert den aktuellen Default Stream für neue Logs.

Die Methode `setDefaultStream` setzt den Default Stream für neue Logs.

Die Methode `getOutputStream` liefert den Asgabestrom, an den die Ausgabe des Logs gesendet wird.

Die Methode `setOutputStream` setzt den Strom, an den Ausgaben des Logs gesandt werden sollen.

Die erste Form der Methode `write` schreibt ein Datenbyte in den Strom.
Falls das Byte nicht ein <nl> ist, wird das Byte dem Datenstrom im Buffer angehängt.
Falls das Byte ein <nl> ist, wird der Buffer in den Stream geschrieben.

Die zweite Form der `write` Methode schreibt ein Byte Array.

Die Methode `toString` liefert den Namen des Logs in Form einer Zeichenkette.

Die Methode `parseLevel` konvertiert eine Zeichenkette mit dem Logging Level in seine interne Darstellung.

1.6.11. Stub und Skeleton Compiler

Der `rmic` Stub und Skeleton Compiler wird benutzt, um passende Stubs und Skeletons für eine spezifische remote Objekt Implementation zu übersetzen. Der Compiler wird mit der voll qualifizierten Klasse des remote Objekts aufgerufen. Die Klasse muss vorgängig erfolgreich übersetzt worden sein.

- der Ort, woe die importierten Klassen stehen, kann man entweder mit Hilfe der CLASSPATH Umgebungsvariable oder aber dem -classpath Argument spezifizieren.
- die von rmic compilierten Klassendateien, werden in das aktuelle Verzeichnis geschrieben, sofern das Argument -d <Klassenzielverzeichnis> nicht angegeben wurde
- das Argument `-keepgenerated` (oder `-keep`) sorgt dafür, dass die generierten Java Quelldateien für Stub und Skeleton nicht gelöscht werden.
- Die Stub Protokoll Version kann mit Hilfe eines Argumentes spezifiziert werden:
  - `-v1.1` kreiert Stubs/Skeletons für die JDK 1.1 Stub Protokoll Version
  - `-vcompat` (Default in JDK 1.2) kreiert Stubs/Skeletons, die sowohl mit JDK 1.1 als auch JDK 1.2 kompatibel sind.
  - `-v1.2` kreiert Stubs für die JDK 1.2 Stub Protokoll Version (Skeletons werden ab JDK1.2 nicht mehr benötigt)
- Die `-show` Option verwendet ein grafisches Unterface für das Programm, wird aber nicht mehr unterstützt
- die meisten `javac` Kommandozeilenargumente sind anwendbar (Ausnahme: `-O`):
  - `-g` generiert Debugging Info
  - `-depend` recompiliert veraltete Dateien rekursiv
  - `-nowarn` unterdrückt Warnungen
  - `-verbose` generiert Meldungen, aus denen ersichtlich wird, was der Compiler gerade tut.
  - `-classpath <Pfad>` specifiziert, wo die Quelldateien und Klassendateien zu finden sind.
  - `-d <directory>` gibt an, wohin die generierten Klassendateien geschrieben werden sollen
  - `-J<runtime flag>` sendet ein Falg an das Java Laufzeitsystem

In JBuilder kann man, wie in der Einführung zu RMI bereits erwähnt, die Dateien markieren, welche bei der Generierung von Stubs und Skeletons berücksichtigt werden sollen.

1.7. RMI - Registry Interfaces

Das RMI System verwendet das `java.rmi.registry.Registry` Interface und die `java.rmi.registry.LocateRegistry` Klasse, als Bootstrap Dienst für das Registrieren und Lesen der Objekte auf Grund ihrer Namen. Eine *Registry* ist ein remote Objekt, mit dessen Hilfe Namen auf entfernte Objekte abgebildet werden. Ein Server Prozess kann seine eigene Registry haben, oder eine Registry kann von mehreren Hosts eingesetzt werden. Die Methoden von `LocateRegistry` werden eingesetzt, um die Registry zu bestimmen, welche auf einem bestimmten Host oder Host und Port (pro Registry kann ein eigener Port angegeben werden). Die Methoden der `java.rmi.Naming` Klasse verwenden die `LocateRegistry.getRegistry` Methode, um auf entfernte Objekte zuzugreifen, welche das `Registry` Interface implementieren.

1.7.1. Das Registry Interface

Das `java.rmi.registry.Registry` remote Interface stellt Methoden zur Verfügung, zum Nachsehen, Binden, Rebinding, Unbinding und Auflisten des Inhalts einer Registry. DIe `java.rmi.Naming` Klasse verwendet das `Registry` remote Interface, um eine URL-basierte Adressierung zur Verfügung zu stellen.

```
package java.rmi.registry;
     public interface Registry extends java.rmi.Remote {
     public static final int REGISTRY_PORT = 1099;
     public java.rmi.Remote lookup(String name)
          throws java.rmi.RemoteException,
          java.rmi.NotBoundException,
          java.rmi.AccessException;
     public void bind(String name, java.rmi.Remote obj)
          throws java.rmi.RemoteException,
          java.rmi.AlreadyBoundException,
          java.rmi.AccessException;
     public void rebind(String name, java.rmi.Remote obj)
          throws java.rmi.RemoteException,
          java.rmi.AccessException;
     public void unbind(String name)
          throws java.rmi.RemoteException,
          java.rmi.NotBoundException,
          java.rmi.AccessException;
     public String[] list()
          throws java.rmi.RemoteException,
          java.rmi.AccessException;
}
```

Der `REGISTRY_PORT` ist der Default Port der Registry.

Die `lookup` Methode liefert das remote Object, welches an den spezifischen Namen *nam*e gebunden ist. Ein remote Objekt implementiert einige remote Interfaces. Clients können das remote Objekt casten, also gemäss einem remite Interface umwandeln (wobei das Casten wie üblich in Java fehlschlagen kann, falls dieses Casten nicht möglich ist).

Die `bind` Methode ordnet einem Namen *name* ein remote Objekt, *ob*j zu. Falls der Namen schon an ein Objekt gebunden ist, wird die `AlreadyBoundExcepton` geworfen.

Die `rebind` Methode assoziiert den Namen *name* mit dem remote Objekt, *ob*j. Eine allfällig bereits bestehende Bindung wird überschrieben

Die `unbind` Methode entfernt die Bindung zwischen dem Namen *name* und dem remote Objekt, *ob*j. Falls der Name noch nicht an ein Objekt gebunden ist, wird die Ausnahme `NotBoundException` geworfen.

Die `list` Methode liefert ein Array von `Strings,` ein Snapshot aller gebundener Namen in der Registry.

Clients können entweder mit Hilfe der `LocateRegistry` und den `Registry` Interfaces oder mit Hilfe der Methoden der URL-based`java.rmi.Naming` Klasse. Die Registry unterstützt `bin`d, `unbin`d, und `rebind` nur von Clients auf dem selben Host wie der Server; eine Abfrage, lookup, kann von irgend einem Host gestartet werden.

1.7.2. Die LocateRegistry Klasse

Die Klasse `java.rmi.registry.LocateRegistry` wird benutzt, um eine Referenz auf die Bootstrap remote Objekt Registry auf einem bestimmten Host (wobei auch der lokale Host zugelassen ist) zu erhalten.

Die Registry implementiert eine eine flache Namenssyntax, welche den Namen eines remote Objektes (eine Zeichenkette) mit einer remote Objekt Referenz herstellt. Diese Bindung zwischen dem Namen und dem remote Objekt geht bei einem Server Restart verloren.

Ein `getRegistry` Aufruf stellt nicht unbedingt eine Verbindung zum remote Host her. Es wird einfach eine lokale Referenz zur entfernten Registry kreiert. Die Registry auf dem entfernten Host muss also nicht mehr unbedingt aktiv sein.

```
package java.rmi.registry;
public final class LocateRegistry {
     public static Registry getRegistry()
          throws java.rmi.RemoteException;
     public static Registry getRegistry(int port)
          throws java.rmi.RemoteException;
     public static Registry getRegistry(String host)
          throws java.rmi.RemoteException;
     public static Registry getRegistry(String host, int port)
          throws java.rmi.RemoteException;
     public static Registry getRegistry(String host, int port,
                RMIClientSocketFactory csf)
          throws RemoteException;
     public static Registry createRegistry(int port)
          throws java.rmi.RemoteException;
     public static Registry createRegistry(int port,
                RMIClientSocketFactory csf,
                RMIServerSocketFactory ssf)
          throws RemoteException;
```

```
}
```

Die vier `getRegistry` Methoden liefern eine Referenz auf eine Registry

- auf dem aktuellen Host an einem speziellen Port,
- auf dem aktuellen Host an einem spezifizierten Port *por*t,
- auf einem spezifizierten Host *host*,
- auf einem spezifizierten Host *host* an einem spezifizierten Port *port*

Die fünfte `getRegistry` Methode (mit einer `RMIClientSocketFactory` als eines ihrer Parameter), liefert einen lokal kreierten remote Stub auf ein entferntes Objekt, einer `Registry` auf einem spezifizierten *host* und *por*t. Die Kommunikation mit der entfernten Registry, deren Stub mit Hilfe dieser Methode konstruiert wurde, wird die `RMIClientSocketFactor`y, *cs*f einsetzen, um eine Socketverbindung mit der Registry auf dem entfernten Host und dem Port herzustellen.

Bemerkung – Eine Registry, welche Rückgabewert der `getRegistry` Methoden ist, ein ein speziell konstruierter Stub, der einen klar definierten Objektidentifier besitzt.
Die Übergabe von Registry Stubs von einer VM an eine andere VM wird nicht unterstützt (zumindest nicht generell: je nach Implementation ist dies jedoch denkbar).

Mit Hilfe der `LocateRegistry.getRegistry` Methoden kann man die einem Host zugeteilte Registry erhalten.

Die `createRegistry` Methoden kreieren und exportieren eine Registry auf dem lokalen Host für den Port *por*t. Die zweite `createRegistry` Methode erlaubt mehr Flexibilität bei der Kommunikation mit der Registry. Ein Aufruf dieser Methode kreiert und exportiert eine `Registry` auf dem lokalen Host, wobei die Registry eine spezielle Socketfactory für die Kommunikation mit dieser Registry verwendet. Die Registry wartet auf ankommende Anfragen an Port *port* unter Verwendung einer `ServerSocket,` welche mit Hilfe der zur Verfügung gestellten `RMIServerSocketFactor`y kreiert wurde. Ein Client wird diesen Socket für die Kommunikation verwenden.

Bemerkung – Falls man eine Registry mit Hilfe der `createRegistry` Methode kreiert wird, bleibt der Server nicht am Leben.

1.7.3. Das RegistryHandler Interface

Bemerkung – Das RegistryHandler Interface wurde in JDK1.2 verworfen. In JDK1.1 wurde dieses Interface nur intern in RMI Implementationen, also nicht für die Anwendungsentwicklung eingesetzt.

```
package java.rmi.registry;
public interface RegistryHandler {
     Registry registryStub(String host, int port)
          throws    java.rmi.RemoteException,
                    java.rmi.UnknownHostException;
     Registry registryImpl(int port)
          throws java.rmi.RemoteException;
}
```

Die Methode `registryStub` liefert einen Stub für die Verbindung mit einer entfernten Registry, mit spezifiziertem Host und Port.Die Methode `registryImpl` konstruiert und exportiert eine Registry am spezifizierten Port. Der Port darf nicht 0 sein.

1.7.4. Aufgabe

Bestimmen Sie in einem RMI Programm die Registry mit Hilfe der getRegistry() Methode

1.8. RMI - Remote Objekt Aktivierung

1.8.1. Übersicht

Verteilte Objektsysteme unterstützen auch langlebige Objekte. Da ein solches System mehrere Tausend Objekte umfassen kann, wäre es unsinnig, alle langlebigen Objekte dauernd aktiv zu lassen. Dadurch würden unnötigerweise viele Systemresourcen blockiert.
Zudem muss die Möglichkeit bestehen, Informationen langfristig zu speichern, so dass die Kommunikation zwischen Objekten zum Beispiel nach einem System Crash wieder aufgebaut werden können.

Objekt Activation ist ein Mechanismus für das Handhaben persistenter (=beständiger) Informationen und die Ausführung von implementierten Objekten.

Mit Hilfe der RMI Aktivierung ist es möglich, Objekte dann zu aktivieren, wenn sie benötigt werden. Das heisst also, dass ein “activatable” remote Objekt beim Aufruf (einer Methode) in die passende Java VM geladen wird, falls es noch nicht vorhanden ist.

1.8.1.1. Terminologie

Ein *aktives* Objekt ist ein remote Objekt, welches instanziert und exportiert wurde, in einer Java VM auf irgend einem System.

Ein *passives* Objekt ist eines, welches noch nicht unstanziert oder exportiert wurde, aber welches aktiviert werden kann

Die Transformation eines passiven in ein aktives Objekt wird als *activation* bezeichnet. Aktivierung bedingt, dass dem Objekt eine VM zugeordnet ist. Diese muss in der Lage sein, das Objekt in die VM zu laden und den Zustand des Objekts zu rekonstruieren, sofern das Objekt bereits existierte.

Im RMI System benutzt man *lazy activatio*n. "Lazy activation" verzögert die Aktivierung eines Objekts, bis der erste Client es benötigt (also eine seiner Methoden aufgerufen wurde).

1.8.1.2. Lazy Activation

Lazy activation eines remote Objekts wird mit Hilfe einer *faulting remote reference* (auch als fault block bezeichnet) realisiert, analog zu "page faults" in Betriebsystemen. Eine "faulting" remote Referenz auf ein remote Objekt “faults in” die aktive Objektreferenz beim ersten Aufruf einer Methode diese Objekts (analog beim Betriebsystem bei der Speicherverwaltung).

Beim Eintreffen eines Faults werden zwei "Handles", Verbindungen, zum remote Objekt aufrecht erhalten: eine persistente (=beständige),eine Aktivierungsidentifikation, und eine transiente (=temporäre). Die remote Aktivierungsidentifikation (*activation ID*) besitzt genug Informationen über das Objekt, um das Objekt aktivieren zu können oder eine andere Stelle zu veranlassen, das Objekt zu aktivieren. Die transiente Referenz ist die aktuelle "live" Referenz auf das Objekt. Falls ein Objekt einen Fault generiert, muss das entfernte Objekt erst aktiviert werden.

Konkret sieht das so aus, dass ein Stub eines remote Objekts einen "faulting" remote Referenztyp besitzt. Dieser besteht konkret aus:

- einer AktivierungsID für das remote Objekt;
- eine "live" Referenz auf ein möglicherweise leeres remote Objekt

Bemerkung – Das RMI System sorgt dafür, dass ein *activatable* oder *unicast* remote Objekt höchstens einmal aktiviert wird. Dies garaniert, dass ein entfernter Methodenaufruf höchstens einmal ausgeführt wird.

1.8.2. Activation Protokoll

Falls in einem remote Aufruf einer Methode die "live" Referenz nicht bekannt ist, sorgt der "faulting" Mechanismus dafür, dass das Objekt aktiviert wird, mit Hilfe eines Activierungsprotokolls (activation protocol).

Das Aktivierungsprotokoll umfasst mehrere Teile:
- die "faulting" Referenz
- den *Activator*
- eine *Activator Gruppe* in der Java VM
- das remote Objekt, welches aktiviert werden soll

Der Activator (einer per Host) überwacht die Aktivierung. Der Activator ist:
- eine Art Datenbank, die Informationen darüber enthält, wie aus der AktivierungsID die nötige Information zum Aktivieren eines Objektes konstruiert werden kann:
  die Objektklasse - ein URL Pfad zum Ort, von dem die Klasse geladen werden kann;
  spezifische Daten, die das Objekt unter Umständen zum Bootstrapping benötigt.
- ein Verwalter von Java VMs, welche auch andere Java VM's starten können und Informationen für die Aktivierung an die korrekte Aktivierungsgruppe in der VM weiter leiten. Zur Performance Optimierung verwaltet der Activator auch einen Cache, in dem die AktivierungsIDs zu aktiven Objekten gespeichert werden.

Die Aktivierungsgruppe (eine pro Java VM) ist die Einheit, die eine Anfrage zur Aktivierung eines Objektes erhält und ein aktives Objekt an den Activator zurück gibt.

Das Aktivierungsprotokoll funktioniert prinzipiell wie folgt:
1. eine "faulting" Referenz verwendet eine AktivierungsID und ruft den Activator mit Hilfe eines RMI internen Interfaces auf, um das zum Identifier gehörende Objekt zu aktivieren.
2. der Activator schaut die Aktivierungsbeschreibung (*activation description*) des zu aktivierenden Objektes nach. Die Objektbeschreibung umfasst:
   - einen Objektgruppenidentifier, mit dessen Hilfe die VM gefunden wird
   - den Klassennamen des Objekts
   - ein URL Pfad zum Ort, von dem der Klassencode des Objekts geladen werden kann.
   - objektspezifische Initialisierungsinformation, in verschlüsselter Form (marshalled). Zum Beispiel könnte der Dateinamen zum Klassencode in diesen Daten stehen.
3. falls die Aktivierungsgruppe, in der das Objekt vorhanden sein sollte, existiert, sendet der Activator die Aktivierungsanfrage an diese Gruppe.
4. falls die Aktivierungsgruppe nicht existiert, dann initialisiert der Activator eine VM und eine Aktivierungsgruppe. Anschliessend wird die Aktivierungsgruppe an diese neu kreierte Gruppe weiter geleitet.

5. Die Aktivierungsgruppe ladet die Klasse für das Objekt und Instanziert das Objekt mit Hilfe eines speziellen Konstruktors, der als Parameter die Aktivierungsbeschreibung verwendet.
6. falls das Objekt aktiviert ist, liefert die Aktivierungsgruppe das Objekt in der verschlüsselten (mashalled) Form an den Activator zurück.
7. der Activator notiert die Aktivierung des Objekts zum Aktivierungsparameter und liefert die aktive Referenz an die "faulting" Referenz zurück.
8. die "faulting" Referenz im Stub liefert schliesslich den Methodenaufruf direkt an das remote Objekt.

Bemerkung – In JDK stellt RMI eine Implementation des Aktivierungssystems als Interfaces. Damit diese eingesetzt werden können, muss das Aktivierungssystem als Daemon gestartet werden : **rmid**.

1.8.3. Implementations Modell für ein “Activatable” Remote Object

Damit ein remote Objekt mit Hilfe einer AktivierungsID zugreifbar ist, muss bei der Entwicklung bereits einiges vorbereitet werden:

- eine Aktivierungsbeschreibung für das remote Objekt muss registriert werden
- es muss ein spezieller Konstruktor in der Objektklasse vorgesehen werden, den das RMI System aufrufen kann, um ein aktivierbares Objekt zu aktivieren.

Eine Aktivierungsbeschreibung(`ActivationDesc`) kann auf unterschiedliche Art und Weise registriert werden:

- mit Hilfe eines Aufrufs der statischen Methode `register` der Klasse `Activatable`,oder
- durch Kreieren eines “activatable” Objekts mit Hilfe des ersten oder zweiten Konstruktors der `Activatable` Klasse

Für ein spezifisches Objekt darf nur die eine oder die andere der beiden oben erwähnten Methoden eingesetzt werden.

1.8.3.1. Die ActivationDesc Klasse

Eine `ActivationDesc` enthält die nötigen Informationen, um ein Objekt zu aktivieren. Insbesondere enthält die Aktivierungsbeschreibung die Identität der Aktivierungsgruppe, den Klassennamen des Objekts, den Codebasis-Pfad (oder die URL), von dem der Objektcode geladen werden kann sowie eine Instanz von `MarshalledObject,` welche objekt-spezifische Informationen enthalten kann, die bei der Aktivierung benötigt werden.

Ein Descriptor wird im Laufe des Aktivierungsprozesses vom Aktivierungssystem konsultiert, um Informationen zu erhalten, mit deren Hilfe das Objekt aktiviert oder neu kreiert werden kann. Als zweites Argument wird das `MarshalledObject` übergeben, damit die Objektdaten entsprechend diesem MarshalledObject gesetzt werden können.

```
package java.rmi.activation;
public final class ActivationDesc
          implements java.io.Serializable{
     public ActivationDesc(String className,
                           String codebase,
```

```
                        java.rmi.MarshalledObject data)
          throws ActivationException;
     public ActivationDesc(String className,
                        String codebase,
     java.rmi.MarshalledObject data,
                        boolean restart)
          throws ActivationException;
     public ActivationDesc(ActivationGroupID groupID,
                        String className,
                        String codebase,
                        java.rmi.MarshalledObject data,
                        boolean restart);
     public ActivationDesc(ActivationGroupID groupID,
                        String className,
                        String codebase,
                        java.rmi.MarshalledObject data);
     public ActivationGroupID getGroupID();
     public String getClassName();
     public String getLocation();
     public java.rmi.MarshalledObject getData()
     public boolean getRestartMode();
}
```

Der erste Konstruktor für `ActivationDesc` konstruiert eine Objektbeschreibung für ein Objekt, dessen Klassennamen *className* ist, das vom *codebase* Pfad geladen werden kann und dessen Initialisierungsinformationen in verschlüsselter Form (marshalled) in *data* gespeichert ist. Als Aktivierungsgruppe wird jene aus der aktuellen Umgebung, gemäss `ActivationGroupID,` verwendet. Falls die aktuelle Gruppe inaktiv ist oder keine Defaultgruppe kreiert werden kann, wird eine `ActivationException` geworfen. Falls die *groupID* null ist, wird eine `IllegalArgumentException` Ausnahme geworfen.

Bemerkung – Als Nebeneffekt wird beim Kreieren einer Aktivierungsbeschreibung `ActivationDesc` eine `ActivationGroup` für diese VM kreiert, falls keine aktiv ist. Die Standard Aktivierungsgruppe verwendet den `java.rmi.RMISecurityManager` als Standard Security Manager.

Der zweite Konstruktor für `ActivationDesc` konstruiert eine Objektbeschreibung analog dem ersten Konstruktor. Der zusätzliche Parameter, *restart*, ist entweder wahr oder falsch. Falls das Objekt einen *restart service* benötigt, also automatisch wieder gestartet wird, wenn der Activator gestartet wird, muss `restart = true` sein. Falls *restart* = false ist, wird das Objekt auf Verlangen aktiviert.

Der dritte Konstruktor für `ActivationDesc` konstruiert eine Objektbeschreibung für ein Objekt der Gruppe *groupID*, und Klassennamen *className*, welche vom *codebase* Pfad geladen werden kann und dessen Initialisierungsinformation in *data* enthalten ist. Alle Objekte mit der selben *groupID* werden in der gleichen Java VM aktiviert.

Der vierte Konstruktor für `ActivationDesc` konstruiert eine Objektbeschreibung, auf die selbe Art und Weise wie der dritte Konstruktor, erlaubt aber einen Restart Modus zu spezifizieren, sofern *restart* true gesetzt wird.

Die `getGroupID` Methode liefert die Identität der Gruppe für das Objekt, welches mit dem Descriptor beschrieben wird. Eine Gruppe stellt eine Möglichkeit dar, Objekte in einer VM logisch zusammen zu fassen.

Die `getClassName` Methode liefert den Klassennamen für das Objekt, welches in der Aktivierungsbeschreibung spezifiziert wird.

Die `getLocation` Methode liefert den Codebase Pfad von dem die Objektklasse (das class File) geladen werden kann.

Die `getData` Methode liefert ein "marshalled Objekt", welches Initialisierungs- (Aktivierungs-) Daten für das im Desciptor beschriebene Objekt enthält.

Die `getRestartMode` Methode liefert true falls der restart Modus enabled ist, für dieses Objekt; sonst liefert die Methode den Wert false..

1.8.3.2. Die ActivationID Klasse

Das Aktivationprotokoll verwendet ActivationID, um remote Objekte zu verwalten, Objekte, die falls nötig aktiviert werden können. Ein ActivationID Objekt, eine Instanz der Klasse `ActivationID`, enthält Informationen zur Aktivierung eines Objektes:

- eine remote Referenz zum Objektactivator und
- eine eindeutige Kennung für das Objekt

.

Ein AktivierungsID für ein Objekt kann man dadurch erhalten, dass man das Objekt beim Aktivierungssystem anmeldet, registriert. Die Registrierung kann auf unterschiedliche Art und Weise geschehen.

- mit Hilfe der `Activatable.register` Methode, oder
- mit Hilfe des ersten oder zweiten `Activatable` Konstruktors , die das Objekt registrieren und exportieren (remote verfügbar machen) oder
- mit Hilfe der ersten oder zweiten Form der `Activatable.exportObject` Methode, die eine AktivierungsID, eine Objektimplementation und den Port als Argument haben. Die Methoden registrieren und exportieren das Objekt.

```
package java.rmi.activation;
public class ActivationID implements java.io.Serializable {
     public ActivationID(Activator activator);
     public Remote activate(boolean force)
          throws ActivationException, UnknownObjectException,
                    java.rmi.RemoteException;
     public boolean equals(Object obj);
     public int hashCode();
}
```

Der Konstruktor für die Klasse `ActivationID` besitzt ein einziges Argumentakes, *activato*r, mit dem die remote Referenz auf einen Activator angegeben wird, mit dessen Hilfe das Objekt, welches zur AktivierungsID gehört, aktiviert werden kann.

Instanzen von `ActivationID` müssen global eindeutig sein.

Die `activate` Methode aktiviert das Objekt, welches zum Activation ID gehört.
Falls die Aktivierung fehlschlägt, wird die Ausnahme `ActivationException` geworfen.
Falls der Objektidentifier nicht bekannt ist, wird die Ausnahme `UnknownObjectExceptio`n geworfen.
Falls der Activatoraufruf fehlschlägt, wird die `RemoteException` geworfen.

Die `equals` Methode implementiert Inhaltsgleichheit: falls alle Felder gleich sind (wertmässig), dann sind die Objekte gleich.

1.8.3.3. Die Activatable Klasse

Die `Activatable` Klasse stellt Support für remote Objekte zur Verfügung, welche persistent sein müssen (beständig) und vom System aktiviert werden können.

Die Klasse `Activatable` ist das wichtigste API zur Implementierung von aktivierbaren Objekten. Bevor Objekte registriert und / oder aktiviert werden können, muss der Daemon **rmid** gestartet werden.

```
package java.rmi.activation;
public abstract class Activatable
     extends java.rmi.server.RemoteServer {
               protected Activatable(String codebase,
               java.rmi.MarshalledObject data,
               boolean restart,
               int port)
          throws ActivationException, java.rmi.RemoteException;
     protected Activatable(String codebase,
               java.rmi.MarshalledObject data,
               boolean restart,
               int port,
               RMIClientSocketFactory csf,
               RMIServerSocketFactory ssf)
          throws ActivationException, java.rmi.RemoteException;
     protected Activatable(ActivationID id, int port)
          throws java.rmi.RemoteException;
     protected Activatable(ActivationID id, int port,
               RMIClientSocketFactory csf,
               RMIServerSocketFactory ssf)
          throws java.rmi.RemoteException;
     protected ActivationID getID();
     public static Remote register(ActivationDesc desc)
          throws UnknownGroupException, ActivationException,
          java.rmi.RemoteException;
     public static boolean inactive(ActivationID id)
          throws UnknownObjectException, ActivationException,
```

```
            java.rmi.RemoteException;
        public static void unregister(ActivationID id)
            throws UnknownObjectException, ActivationException,
            java.rmi.RemoteException;
        public static ActivationID exportObject(Remote obj,
                  String codebase,
                  MarshalledObject data,
                  boolean restart,
                  int port)
            throws ActivationException, java.rmi.RemoteException;
        public static ActivationID exportObject(Remote obj,
                  String codebase,
                  MarshalledObject data,
                  boolean restart,
                  int port,
                  RMIClientSocketFactory csf,
                  RMIServerSocketFactory ssf)
            throws ActivationException, java.rmi.RemoteException;
        public static Remote exportObject(Remote obj,
                  ActivationID id,
                  int port)
                  throws java.rmi.RemoteException;
                  public static Remote exportObject(Remote obj,
                  ActivationID id,
                  int port,
                  RMIClientSocketFactory csf,
                  RMIServerSocketFactory ssf)
            throws java.rmi.RemoteException;
        public static boolean unexportObject(Remote obj,
                  boolean force)
            throws java.rmi.NoSuchObjectException;
}
```

Eine Implementation für ein aktivierbares remote Objekt kann, muss aber nicht die Klasse `Activatable` erweitern. Eine solche Implementation erbt die entsprechenden Definitionen für `hashCode` und `equals` Methoden von der Oberklasse `java.rmi.server.RemoteObject`.

Der erste Konstruktor für die `Activatable` Klasse wird eingesetzt zur Registrierung und den Export der Objekts für den Port *port* (falls *port* fehlt, wird ein Defaultport agenommen). Der Objektpfad (URL) zum Laden der Klasse ist in *codbas*e beschrieben, die Initialisierungsdaten stehen in *data*.I*frestart* ist true, falls das Objekt nach einen Absturz automatisch wieder gestartet wird. Falls *restart* false ist, wird das Objekt auf Verlangen gestartet).
Konkrete Unterklassen der abstrakten `Activatable` Klasse müssen den ersten Konstruktor aufrufen, um das Objekt zu registrieren und zu exportieren.
Der Konstruktor wirft die Ausnahme `ActivationException` falls die Registrierung des Objekts fehlschlägt. `RemoteException` wird geworfen, falls der Export des Objekts an das RMI Laufzeitsystem nicht glückt.

Der zweite Konstruktor ist der selbe wie der erste, erlaubt aber die Spezifikation der Client und Server Socket Factory, mit deren Objekte (Sockets) kommuniziert werden soll.. Details stehen im Abschnitt “RMI Socket Factories”.

Der dritte Konstruktor aktiviert und exportiert das Objekt (mit der `ActivationID`, *i*d) an Port *por*t. Eine konkrete Unterklasse der `Activatable` Klasse muss diesen Konstruktor mit den Parametern aufrufen:

- die AktivierungsID des Objekts (`ActivationID`), und
- die Initialisierungdaten (ein `MarshalledObjec`t).

Zusätzlich zur Konstruktion des Objekts wird das Objekt auch noch exportiert, also dem RMI Laufzeitsystem zur Verfügung gestellt, um auf eintreffende Aufrufe antworten zu können. Falls ein fehler beim Exportieren auftritt, wird die Ausnahme `RemoteException` geworfen.

Der vierte Konstruktor unterschiedet sich kaum vom dritten. Zusätzlich wird wieder die Möglichkeit geschaffen, SocketFactories anzugeben.

Die `getID` Methode liefert die AktivierungsID des Objekts. Diese Methode ist protected; damit können nur Unterklassen die ObjektID erfragen. Mit Hilfe der AktivierungsID kann man feststellen, ob ein Objekt inaktiv ist, oder man kann das Objekt aus der Registry löschen.

Die `register` Methode registriert, mit Hilfe des Aktivierungssystems, eine Obektbeschreibung, *des*c, zur Aktivierung entfernter Objekte, damit ein remote Objekt aktiviert werden kann. Diese Methode kann ein Objekt registrieren, ohne das Objekt zuerst zu kreieren. Die Methode liefert auch den `Remote` Stub für das aktivierbare Objekt, so dass das Objekt später kreiert und aktiviert werden kann.
Die Methode wirft die `UnknownGroupException` falls der Gruppenidentifier in *desc* im Aktivierungssystem nicht registriert ist.
`ActivationException` wird geworfen, falls das Aktivierungssystem nicht läuft.
Schliesslich wird die `RemoteException` geworfen, falls der remote Methodenaufruf nicht erfolgreich ist.

Die `inactive` Methode wird eingesetzt, um dem System mitzuteilen, dass das Objekt mit der AktivierungsID *id* zur Zeit inaktiv ist.
Falls das Objekt zur Zeit aktiv ist, wird das Objekt aus dem RMI Laufzeitsystem entfernt, sofern keine Anfragen pendent sind. Das Objekt wird also keine neuen Anfragen empfangen können.
Zudem wird die `ActivationGroup` der VM informiert, dass das Objekt inaktiv ist. Die Gruppe informiert ihren `ActivationMonito`r darüber.

Falls der Aufruf erfolgreich beendet werden kann, hat der Activator die Möglichkeit später das Objekt erneut zu aktivieren.Die `inactive` Methode liefert bei erfolgreichem Abschluss den Wert *true, false* falls das Objekt nicht aus dem RMI Laufzeitsystem entfernt werden konnte.
Die Methode wirft die Ausnahme `UnknownObjectException,` falls das Objekt unbekannt ist;
eine `ActivationException` wird geworfen, falls die Gruppe inaktiv ist;
eine `RemoteException` falls die Benachrichtigung des ActivationMonitos missglückte.

Die `unregister` Methode macht eine vorgängige Regsitrierung rückgängig.
Beim Versuch das Objekt erneut zu aktivieren, würde eine `UnknownObjectException` geworfen;
falls das Aktivierungssystem nicht läuft, wird eine `ActivationException` geworfen;
falls ein remote Call an das Aktivierungssystem misglückt, wird die `RemoteException` geworfen.

Die `exportObject` Methode kann von einem "activatable"
Objekt aufgerufen werden, welches die `Activatable` Klasse nicht erweitert, um
a) die Objektaktivierungsbeschreibung *des*c, zu registrieren. desc wird mit Hilfe der Codebase *codebase* und *data* und dem Aktivierungssystem konstruiert
b) das remote Objekt *obj* wird exportiert, an Port *port* (falls dieser nicht null ist; sonst wird ein anonymer Port verwendet)

Nachdem das Objekt exportiert ist, kann es ankommende RMI Anfragen behandeln.

Diese `exportObject` Method liefert den AktivierungsID, welche mit Hilfe der Registrierung des Desciptors *desc* und dem Aktivierungssystem erhalten wird.
Falls die Aktivierungsgruppe in der VM nicht aktiv ist, wird die `ActivationException` Ausnahme geworfen;
falls die Objektregistrierung fehlschlägt, oder der Export, dann wird die `RemoteException` geworfen.
Die Methode muss nicht aufgerufen werden, falls *obj* die Klasse `Activatable` erweitert, da der `Activatable` Konstruktor diese Methode aufruft.

Die zweite `exportObject` Methode ist analog zur ersten, ausser dass die Client und Server Socket Factories angegeben werden können.

Die weiteren Formen der `exportObject` Methode erlauben detaillierte Kontrollen über den Exportprozess.

Die `unexportObject` Method modifiziert das remote Object, *ob*j, so dass es für eintreffende Anfragen nicht mehr zur Verfügung steht.
Der force Parameter kontrolliert die Art und Weise,wie mit pendenten Anfragen umgegangen wird.

1.8.3.3.1. Konstruktion eines Activatable Remote Objekts

Damit ein Objekt aktiviert werden kann, muss die "activatable" Objekt Implementations Klasse (die eventuell die `Activatable` Klasse erweitert) einen speziellen public Konstruktor mit zwei Argumenten definieren. Die Argumente sind: der Aktivierungdidentifizierer, vom Typ `ActivationID`, und die Aktivierungsdaten in Form eines `java.rmi.MarshalledObject` Objekts.

Die Aktivierungsgruppe aktiviert ein remote Objekt innerhalb der VM mit Hilfe des speziellen Konstruktor (den wir gleich beschreiben werden).

Das folgende Beispiel veranschaulicht den Vorgang:

```
package examples;
public interface Server extends java.rmi.Remote {
public void doImportantStuff()
throws java.rmi.RemoteException;
}
public class ServerImpl extends Activatable implements Server
{
// Konstruktor für initiale Konstruktion, Registration und
// Export
public ServerImpl(String codebase, MarshalledObject data)
throws ActivationException, java.rmi.RemoteException
{
// registrieren des Objekts, mit dem Aktivation System
// und Export an einen anonymous Port
     super(codebase, data, false, 0);
}
// Konstruktor für die Aktivierung und den Export;

public ServerImpl(ActivationID id, MarshalledObject data)
throws java.rmi.RemoteException
{
// Superklasse Konstruktor :exportiert
     super(id, 0);
// initialisieren der Objekte
}
     public void doImportantStuff() { ... }
}
```

Un hier noch ein Beispiel für einen Server, der `Activatable` nicht erweitert, dafür aber die Klasse `ServerImpl`

```
package examples;
public class ServerImpl extends SomeClass implements Server
{
//
public ServerImpl(String codebase, MarshalledObject data)
throws ActivationException, java.rmi.RemoteException
{
// register and export the object
Activatable.exportObject(this, codebase, data, false, 0);
}
// Aktivierung
public ServerImpl(ActivationID id, MarshalledObject data)
throws java.rmi.RemoteException
{
// Objektexport
Activatable.exportObject(this, id, 0);
}
```

```
public void doImportantStuff() { ... }
}
```

1.8.3.3.2. Registrierung eines Activation Descriptor ohne ein Objekt zu kreieren

Hier eine Skizze eines solchen Programms:

```
Server server;
ActivationDesc desc;
String codebase = “http://meinHost/codebase/”;
MarshalledObject data = new MarshalledObject(“einige Daten”);
desc = new ActivationDesc(“beispiel.ServerImpl”, codebase,
data);
server = (Server)Activatable.register(desc);
```

Der `register` Aufruf liefert einen `Remote` Stub, den Stub für das `beispiele.ServerImpl` Objekt und implementiert die selben remote Interfaces, wie `beispiele.ServerImpl`.

1.8.4. Activation Interfaces

Im RMI Aktivierungsprotokoll muss der Activator dafür sorgen, dass das System funktionsfähig bleibt:

- wie alle Systemdaemons muss der Daemon aktiv bleiben, solange das System läuft und
- der Activator muss dafür sorgen, dass remote Objekte nicht aktiviert werdem, sofern das Objekt bereits aktiv ist

Der Activator unterhält eine entsprechende Datenbank mit den relevanten Informationen über die Gruppen und Objekte, die an der Aktivierung beteiligt sind.

1.8.4.1. Das Activator Interface

Der Activator ist eine der Programmteile, die am Aktivierungsprozess beteiligt sind. Falls ein "Fault" auftritt (in einem Stub), wird die `activate` Methode des Activators aufgerufen, um eine "live" Referenz zu einem aktivierbaren remote Objekt zu erhalten.

Der Ablauf sieht folgendermassen aus:

1. der Activator liest die Aktivierungsbeschreibung zur AktivierungsID
2. bestimmt die Gruppe, in der das Objekt aktiviert werden soll
3. ruft die Methode newInstance auf
4. und startet diese Gruppe

Der Activator muss aber auch die korrekte Ausführung überwachen und im Fehlerfall Listen nachführen (nicht mehr erreichbare remote Referenz).

```
package java.rmi.activation;
public interface Activator extends java.rmi.Remote{
```

```
     java.rmi.MarshalledObject activate(ActivationID id,
          boolean force)
          throws UnknownObjectException, ActivationException,
          java.rmi.RemoteException;
}
```

Die `activate` Methode aktiviert das Objekt mit der AktivierungsID id. Die Behandlung des force Parameters haben wir bereits besprochen

1.8.4.2. Das ActivationSystem Interface

Das `ActivationSystem` stellt Mechanismen zur Verfügung, mit deren Hilfe Gruppen und *activatable* Objekte, die in diesen Gruppen aktiviert werden sollen, regsitriert werden können. Das `ActivationSystem` arbeitet eng mit dem `Activator` zusammen. Dieser aktiviert Objekte mit Hilfe der Klassen / Interfaces `ActivationSystem` und `ActivationMonitor`, der Informationen über aktive und inaktive Gruppen und Objekte erhält.

```
package java.rmi.activation;
public interface ActivationSystem extends java.rmi.Remote
{
     public static final int SYSTEM_PORT = 1098;
     ActivationGroupID registerGroup(ActivationGroupDesc desc)
          throws ActivationException, java.rmi.RemoteException;
     ActivationMonitor activeGroup(ActivationGroupID id,
               ActivationInstantiator group,
               long incarnation)
          throws UnknownGroupException, ActivationException,
          java.rmi.RemoteException;
     void unregisterGroup(ActivationGroupID id)
          throws ActivationException, UnknownGroupException,
          java.rmi.RemoteException;
     ActivationID registerObject(ActivationDesc desc)
          throws ActivationException, UnknownGroupException,
          java.rmi.RemoteException;
     void unregisterObject(ActivationID id)
          throws ActivationException, UnknownObjectException,
          java.rmi.RemoteException;
     void shutdown() throws java.rmi.RemoteException;
}
```

Bemerkung – Zur Sicherheit wird eine `java.rmi.AccessException` Ausnahme, eine Unterklasse von `java.rmi.RemoteException` geworfen, falls eine der obigen Methoden von einem Client aufgerufen wird, der nicht auf dem selben Host ist wie das Aktivierungssystem.

Die `registerObject` Methode registriert eine Aktivitätsbeschreibung und erhält eine AktivitätenID für ein aktivierbares remote Objekt.
Die Ausnahmen, die geworfen werden können,sind aus der obigen Liste ersichtlich.

Die `unregisterObject` Methode entfernt eine AktivierungsID und die dazu gehörige Beschreibung, die mit dem `ActivationSystem` registriert wurden.

Die `registerGroup` Method registriert eine Aktivierungsgruppe und das Aktivierungssystem liefert die `ActivationGroupID` dieser Gruppe zurück.

Die `activeGroup` Methode ist eine Callback Methode von `ActivationGroup.` Sie informiert das Aktivationsystem, dass die Gruppe *group* nun aktiv ist.

Die `unregisterGroup` Method entfernt eine Aktivationgruppe.

Die `shutdown` Methode fährt das Aktivationsystem sanft herunter.
Dies entspricht einem Aufruf des Kommandos:
`rmid -stop [-port num]`

1.8.4.3. Die ActivationMonitor Klasse

Ein `ActivationMonitor` wird pro `ActivationGroup` spezifiziert. Man erhält die aktive Monitor Gruppen-Klasse mit Hilfe des Methodenaufrufs. Die Gruppe muss den Monitor über Ereignisse informieren.

Beispielsweise
falls ein Objekt aktiv, eine Gruppe inaktiv ... wird.

```
package java.rmi.activation;
public interface ActivationMonitor
     extends java.rmi.Remote
{
     public abstract void inactiveObject(ActivationID id)
          throws UnknownObjectException, RemoteException;
     public void activeObject(ActivationID id,
               java.rmi.MarshalledObject mobj)
          throws UnknownObjectException,
           java.rmi.RemoteException;
     public void inactiveGroup(ActivationGroupID id,
               long incarnation)
          throws UnknownGroupException,
          java.rmi.RemoteException;
}
```

Eine Aktivierungsgruppe ruft die Monitor Methode `inactiveObject` auf, falls ein Objekt in der Gruppe deaktiviert wird.

Ein Methodenaufruf `activeObject` informiert den `ActivationMonitor,` dass das Objekt mit der Identität *id* nun aktiv ist

Die `inactiveGroup` informiert den Monitor, dass die Gruppe nun inaktiv ist.

1.8.4.4. Die ActivationInstantiator Klasse

Die `ActivationInstantiator` Klasse ist verantworlich für das Kreieren von Instanzen von activatable Objekten.

```
package java.rmi.activation;
public interface ActivationInstantiator
     extends java.rmi.Remote
{
     public MarshalledObject newInstance(ActivationID id,
               ActivationDesc desc)
          throws ActivationException, java.rmi.RemoteException;
}
```

Damit wird ein Objekt in der Gruppe mit den angegebenen Parametern zu kreieren.

1.8.4.5. Die ActivationGroupDesc Klasse

Eine Activation Group Beschreibung (`ActivationGroupDesc`) enthält Informationen, die benötigt werden, um eine Activation Group neu zu kreieren, in der selben Java VM.

Eine solche Beschreibung umfasst:

- den Klassennamen der Gruppe
- den Codebase the group's codebase path (the location of the group's class), and
- ein “marshalled” Objekt mit den objektspezifischen Daten.

Die Klasse der Gruppe muss eine Subklasse der `ActivationGroup` sein.
Der Konstruktor benutzt zwei Argumente:

- die `ActivationGroupID`, und
- Initialiierungsdaten (in einem `java.rmi.MarshalledObject`)

```
package java.rmi.activation;
public final class ActivationGroupDesc
          implements java.io.Serializable{
     public ActivationGroupDesc(java.util.Properties props,
          CommandEnvironment env);
     public ActivationGroupDesc(String className,
          String codebase,
          java.rmi.MarshalledObject data,
          java.util.Properties props,
          CommandEnvironment env);
     public String getClassName();
     public String getLocation();
     public java.rmi.MarshalledObject getData();
     public CommandEnvironment getCommandEnvironment();
     public java.util.Properties getPropertiesOverrides();
}
```

Die `getClassName` Methode liefert den Klassennamen der Gruppe.

Die andern Methoden benötigen kaum eine Erklärung.

1.8.4.6. Die ActivationGroupDesc.CommandEnvironment Klasse

Mit dieser Klasse lassen sich Systemeigenschaften überschreiben. Diese betreffen jeweils eine `ActivationGroup`.

```
public static class CommandEnvironment implements
java.io.Serializable{
     public CommandEnvironment(String cmdpath, String[] args);
     public boolean equals(java.lang.Object);
     public String[] getCommandOptions();
     public String getCommandPath();
     public int hashCode();
}
```

Der Konstruktor kreiert ein CommandEnvironment mit gegebenem Commandpfad, *cmdpath*, und weiteren Parametern, *args*.

Die `getCommandOptions` liefert die Option.

Die Methode `getCommandPath` liefert die Kommandozeile.

1.8.4.7. Die ActivationGroupID Klasse

Die Identifier für eine registrierte Activation Gruppe dient verschiedenen Zielen:

- sie identifizieren die Gruppe eindeutig innerhalb eines Aktivierungssystems
- sie enthält eine Referenz zum Aktivierungssystem der Gruppe

```
package java.rmi.activation;
public class ActivationGroupID implements java.io.Serializable
{
     public ActivationGroupID(ActivationSystem system);
     public ActivationSystem getSystem();
     public boolean equals(Object obj);
     public int hashCode();
}
```

Der `ActivationGroupID` Konstruktor kreiert einen eindeutigen Gruppenidentifier zum Aktivierungssystem *system.*

Die `getSystem` Methode liefert das Aktivierungssystem für die Gruppe.

1.8.4.8. Die ActivationGroup Klasse

Eine `ActivationGroup` ist verantwortlich für das Kreieren neuer Instanzen eines "activatable" Objekts in dieser Gruppe, wobei auch der `ActivationMonitor` informiert wird, falls ein Objekt aktiv oder inaktiv wird, oder eine ganze Gruppe aktiv oder inaktiv wird.

Eine `ActivationGroup` wird *ursprünglich* auf eine der folgenden Art und Weise kreiert:

- als Nebeneffekt beim Kreieren einer `ActivationDesc` für ein Objekt, oder

- durch expliziten Aufruf der Methode `ActivationGroup.createGroup` oder
- als Seiteneffekt bei der Aktivierung des ersten Objekts einer Gruppe, deren `ActivationGroupDesc` lediglich registriert war.

Die `createGroup` Methode verlangt von der Gruppe, die sie kreieren muss, folgendes:

1. die Gruppe muss eine konkrete Unterklasse von `ActivationGroup` sein
2. der Konstruktor muss zwei Argumente besitzen
   - die `ActivationGroupID`, und
   - Initialisierungsdaten, ein `MarshalledObject`

```
package java.rmi.activation;
public abstract class ActivationGroup extends
          UnicastRemoteObject
          implements ActivationInstantiator{
     protected ActivationGroup(ActivationGroupID groupID)
          throws java.rmi.RemoteException;
     public abstract MarshalledObject newInstance(ActivationID
                    id, ActivationDesc desc)
          throws ActivationException, java.rmi.RemoteException;
     public abstract boolean inactiveObject(ActivationID id)
          throws ActivationException, UnknownObjectException,
          java.rmi.RemoteException;
     public static ActivationGroup createGroup(
                    ActivationGroupID id,
                    ActivationGroupDesc desc,
                    long incarnation)
          throws ActivationException;
     public static ActivationGroupID currentGroupID();
     public static void setSystem(ActivationSystem system)
          throws ActivationException;
     public static ActivationSystem getSystem()
          throws ActivationException;
     protected void activeObject(ActivationID id,
               java.rmi.MarshalledObject mobj)
          throws ActivationException, UnknownObjectException,
          java.rmi.RemoteException;
     protected void inactiveGroup()
          throws UnknownGroupException,
          java.rmi.RemoteException;
}
```

1.8.4.9. Die MarshalledObject Klasse

Ein `MarshalledObject` ist ein Container für ein Objekt. Dadurch kann das Objekt als Parameter in einem RMI Calls eingesetzt werden. Jedes remote Objekt im `MarshalledObject` wird als serialisierte Instanz seines Stubs dargestellt.

```
package java.rmi;
public final class MarshalledObject implements
java.io.Serializable
```

```
     {
     public MarshalledObject(Object obj)
          throws java.io.IOException;
     public Object get()
          throws java.io.IOException, ClassNotFoundException;
     public int hashCode();
     public boolean equals();
}
```

Der `MarshalledObject` Konstruktor verwendet ein serialisiertes Objekt *obj*, als einziges Argument.

Die Bedeutung der einzelnen Methoden dürfte klar sein, auf Grund der Namen und Parameter der Methoden

1.9. RMI - Stub / Skeleton Interfaces

In diesem Abschnitt befassen wir uns mit den Interfaces und Klassen, die von den Stubs und Skeletons benötigt werden und vom `rmic` Stub Compiler generiert werden. Ab JDK 1.2 braucht man die Skeletons nicht mehr, also erscheint im JBuilder die "deprecated" Meldung.

Die `java.rmi.server.RemoteStub` Klasse ist die gemeinsame Oberklasse für Stubs von remote Objekten. Stub Objekte sind umfassen genau die gleichen remote Interfaces, wie die aktuelle Implementation des remote Objektes

.

```
package java.rmi.server;
public abstract class RemoteStub extends java.rmi.RemoteObject
{
     protected RemoteStub();
     protected RemoteStub(RemoteRef ref);
     protected static void setRef(RemoteStub stub,
                                  RemoteRef ref);
}
```

Der erste Konstruktor kreiert einen Stub mit einer null remote Referenz.

Der zweite Konstruktor kreiert einen Stub zu einer gegebenen Referenz, *ref*.

Die `setRef` Methode wurde in JDK1.2 verworfen.

1.9.1. Typen Äquivalenz von Remote Objekten und Stub Klassen

Clients kommunizieren mit Stub (Stellvertreter, surrogate) Objekten, die *genau* die gleichen remote Interfaces definieren, wie die definierte remote Klasse. Der Stub umfasst keine der lokalen Interfaces, Methoden und Datenfelder. Das liegt daran, dass der Stub aus der detailliertesten Darstellung des remote Objekts kreiert wird.

Beispiel:
falls C die Klasse B erweitert, B die Klasse A erweitert, aber nur B remote Interfaces implementiert, dann wird der Stub aus dem Class File der Klasse B generiert, also nicht C.

Aus der Sicht des Java Systems besitzt der Stub das selbe remote Verhalten wie der Server. Daher kann der Client alle Java spezifischen Techniken, wie Castung, auf den Stub anwenden.

Stubs werden mit Hilfe des **rmic** Compilers generiert.

1.9.2. Semantik von final deklarieten Object Method

Die folgenden Methoden sind als `final` in der `java.lang.Object` Klasse deklariert und können somit durch keine Implementation überschrieben werden:

- `getClass`
- notify
- notifyAll
- `wait`

Die Standardimplementation der Methode `getClass` kann auf alle Java Objekte angewandt werden, lokale und entfernte. Deswegen benötigen wir keine neue Form der Methode in einem verteilten Umfeld.

Die `wait` und `notify` Methoden der `java.lang.Object` Klasse behandelt die Synchronisation von Threads. Die Bedeutung in verteilten Systemen ist jedoch leicht anders, wegen der unterschieldichen Threadbehandlung in verteilten Systemen: diese Methoden agieren auf den Stubs, nicht auf den entfernten Objekten.

1.9.3. Das RemoteCall Interface

Das Interface `RemoteCall` ist eine Abstraktion, welche bei Stubs und Skeletons von remote Objekten eingesetzt werden, um Methodenaufrufe ausführen zu können.

Bemerkungen – Das `RemoteCall` Interface wurde in JDK 1.2. verworfen. Anstelle diese Interfaces wird nun die `invoke` Methode eingesetzt.

```
package java.rmi.server;
import java.io.*;
public interface RemoteCall {
     ObjectOutput getOutputStream() throws IOException;
     void releaseOutputStream() throws IOException;
     ObjectInput getInputStream() throws IOException;
     void releaseInputStream() throws IOException;
     ObjectOutput getResultStream(boolean success)
          throws IOException, StreamCorruptedException;
     void executeCall() throws Exception;
     void done() throws IOException;
}
```

Die Methode `getOutputStream` liefert den Output Stream, in den marshalled Argumente oder Rückgabewerte geschrieben werden.

Die Methode `releaseOutputStream` gibt den Output Stream frei.

Die Methode `getInputStream` liefert den InputStream aus dem der Stub die Ergebnisse oder Skeleton unmarshalled Parameter lesen kann
.
Die Methode `releaseInputStream` gibt den Input Stream frei.

Die Methode `getResultStream` liefert einen Output Stream. Die Ergebnisse sollten pro remote Methodenaufruf nur einmal lesbar sein. Falls das Ergebnis bereits erfolgreich gelesen wurde, wird bei einem erneuten Aufruf die `StreamCorruptedException` geworfen.

Die Methode `executeCall` versucht einen Aufruf auszuführen.

Die Methode `done` gestattet eine Art "Aufräumen" nach einem Methodenaufruf.
.

1.9.4. Das RemoteRef Interface

Das Interface `RemoteRef` stellt eine Verbindung, eine Handle / Hantel zum entfernten Objekt dar.

Jeder Stub enthält eine Instanz von RemoteRef, welche die konkrete Darstellung einer Referenz enthält. Mit Hilfe dieser Darstellung / Referenz werden die Aufrufe der netfernten Methoden ausgeführt.

```
package java.rmi.server;
public interface RemoteRef extends java.io.Externalizable {
     Object invoke(Remote obj,
                   java.lang.reflect.Method method,
                   Object[] params,
                   long opnum)
          throws Exception;
     RemoteCall newCall(RemoteObject obj, Operation[] op,
                   int opnum,
                   long hash)
          throws RemoteException;
     void invoke(RemoteCall call) throws Exception;
     void done(RemoteCall call) throws RemoteException;
     String getRefClass(java.io.ObjectOutput out);
     int remoteHashCode();
     boolean remoteEquals(RemoteRef obj);
     String remoteToString();
}
```

Die `invoke` Methode delegiert den Methodenaufruf an die remote Referenz des Stubs *(ob*j). Die remote Referenz kümmert sich um den Aufbau der Verbindung zum remote Host, der Umwandlung (marshalling) der Methodenaufrufe *method,* und der Parameter *param*s. Diese Methode liefert entweder das Ergebnis des Methodenaufrufs auf dem entfernten Host oder sie wirft eine Ausnahme `RemoteException.`

Der Parameter *opnu*m, stellt eine Hashcodierung der Methodensignatur dar, zumindest kann der Parameter so eingesetzt werden.

opnum ist a 64-bit (long) Integer stellen einen Hashcode gemäss National Institute of Standards and Technology (NIST) Secure Hash Algorithm (SHA-1) dar.

Bemerkung – Die Methoden `newCal`l, `invoke` und `done` wurden in JDK 1.2 verworfen.

Die Stubs, welche mit dem rmic ab Version JDK1.2 generiert werden, verwenden diese Methoden nicht mehr. Sie wurden durch die `invoke` Methode ersetzt.

1.9.5. Das ServerRef Interface

Das Interface `ServerRef` stellt die Server-seitige Hantel / Referenz für das remote Objekt dar.

```
package java.rmi.server;
public interface ServerRef extends RemoteRef {
     RemoteStub exportObject(java.rmi.Remote obj, Object data)
          throws java.rmi.RemoteException;
     String getClientHost() throws ServerNotActiveException;
}
```

Die Methode `exportObject` findet oder kreiert ein Client Stub Objekt für die `Remote` Objekt Implementation *obj*.Der Parameter *data* enthält nötige Informationen für den Export des Objekts (zum Beispiel die Port Nummer).

Die Methode `getClientHost` liefert den Host Namen des aktuellen Clients.
Falls ein remote Methoden Aufruf schief geht, wird die `ServerNotActiveException` geworfen.

1.9.6. Das Skeleton Interface

Das Interface `Skeleton` wird einzig und allein von Implementationen der Skeletons benutzt, die vom **rmic** generiert werden.

Ein Skeleton für ein remote Objekt ist eine serverseitige Grösse, die alle Aufrufe an die aktuelle Objektimplementation abliefert.

Hinweis – Das `Skeleton` Interfaces wurde in JDK1.2 verworfen. Mit `rmic -vcompa`t, kann man Stubs und Skeletons generieren, die kompatibel für JDK 1.2 und 1.1 sind.

```
package java.rmi.server;
public interface Skeleton {
     void dispatch(Remote obj, RemoteCall call, int opnum,
                   long   hash)
          throws Exception;
     Operation[] getOperations();
}
```

Die `dispatch` Methode entschlüsselt, unmarshals alle Argumente aus dem Input Stream eines *call* Objekts, ruft die Methode auf (spezifiziert mit dem *opnu*m Code) mitHilfe der aktuellen remote Objekt Implementation *obj*, und marshals / verschlüsselt die Rückgabewerte oder wirft eine Ausnahme, falls ein Fehler auftritt.

Die `getOperations` Method liefert ein Array mit einer Beschreibung der Operationen / Methoden des remote Objektes.

1.9.7. Die Operation Klasse

Die Klasse `Operation` enthält eine Beschreibung einer Java Methode für ein remote Objekt.

Bemerkung – Das `Operation` Interface wurde ab JDK 1.2 verworfen. Das JDK 1.2 Stub Protokoll verwendet die veraltete `RemoteRef.invoke` Method nicht mehr.
Auch hier wird die neu definierte Methode `invoke` eingesetzt.

```
package java.rmi.server;
public class Operation {
     public Operation(String op);
     public String getOperation();
     public String toString();
}
```

1.10. RMI - Garbage Collector Interfaces

1.10.1. Das Interface DGC

Die Abstraktion / das Interface des DBG wird für die serverseitige Garbage Collection Algorithmen zuständig. Das Interface enthält zwei Methoden:

1. `dirty` und
2. `clean`.

Die `dirty` Methode wird dann aufgerufen, wenn eine entfernte Referenz in einem Client entpackt (unmarshalled) wird. Der Client wird mit Hilfe seiner VMID identifiziert. Ein dazu gehörender `clean` Methodenaufruf wird dann abgesetzt, falls auf dem Client keine weitere Refernz auf das entfernte Objekt mehr existiert.

Falls ein dirty Call nicht erfolgreich ist, muss ein verstärkter Clean Aufruf abgesetzt werden, sonst gibt es Probleme mit der Sequenznummer der Aufrufe (jeder Aufruf erhält intern eine Sequenznummer, analog den Transaktionsnummern bei den Datenbanken).

Die Referenz auf ein entferntes Objekt wird vom Client für eine bestimmte Zeitdauer *geleased*. Die Leaseperiode startet beim Aufruf des `dirty` Calls, also nach dem Auspacken des Objektes bzw. der Referenz beim Client. Es liegt am Client das Leasing aufrecht zu erhalten, indem er zusätzliche (`dirty`) Methodenaufrufe tätigt, bevor die Leasingdauer abläuft. Falls der Client das Leasing nicht erneuert bevor die Leasingdauer abläuft, nimmt der Distributed Garbage Collector an, dass das remote Objekt vom Client nicht länger benötigt wird.

```
package java.rmi.dgc;
import java.rmi.server.ObjID;
public interface DGC extends java.rmi.Remote {
     Lease dirty(ObjID[] ids, long sequenceNum, Lease lease)
               throws java.rmi.RemoteException;
```

```
    void clean(ObjID[] ids, long seqNum, VMID vmid,
                    boolean strong)
              throws java.rmi.RemoteException;
}
```

Die Methode `dirty` verlangt ein Leasing für das entfernte Objekt, welches mit Hilfe seiner `ObjID` *`ids`* spezifiziert wird. Das Leasingobjekt (`Lease` Klasse des DGC Paketes) *`lease`* enthält die eindeutige ID der Virtuellen Maschine (VMID) und eine bestimmte (angeforderte) Leasingdauer.

Pro entferntes Objekt unterhält die lokale Virtuelle Maschine (jene, von der das Objekt stammt), eine *Referenzliste* - eine Liste aller Clients, welche eine Referenz auf das Objekt erhielten.

Falls das Leasing bewilligt wird, trägt der Garbage Collector die VMID des Clients in die Referenzliste ein für jedes remote Objekt *ids* ein. Die Sequenznummer, der *sequenceNum* Parameter, ist eine Sequenznummer, mit deren Hilfe der Garbage Collector Aufrufe bestimmen kann. Die Sequenznummer sollte bei jedem Aufruf erhöht werden.

Einige Clients sind nicht in der Lage eine eindeutige VMID zu generieren. Der Grund ist der, dass eine VMID nur dann eine universelle eindeutige Identifikation ist, falls der Client auf seine (Gast) Hostadresse zugreifen kann. Dies ist unter Umständen aber nicht möglich, zum Beispiel wegen Sicherheitseinschränkungen.

In diesem Fall verwendet der Client als VMID `null` und der Distributed Garbage Collector ordnet dem Client eine VMID zu.

Der `dirty` Call liefert ein `Lease` Objekt, welches die benutzte VMID enthält, sowie die Leasingperiode, die für das remote Objekt gewährt wurde. (Ein Server kann von sich aus entscheiden, dass er eine reduzierte Leasingdauer gewährt). Ein Client muss, nachdem das remote Objekt benutzt hat, bei einem späteren clean Aufruf an den DGC die VMID benutzen.

Pro remote Objekt in der Virtuellen Maschine muss ein Client nur einen `dirty` Aufruf tätigen, selbst wenn das remote Objekt im Client mehrfach referenziert wird.

Der Client kann die Leasingdauer verlängern, indem er einen weiteren `dirty` Call absetzt, bevor die Leasingdauer abgelaufen ist.

Falls der Client das remote Objekt nicht mehr benötigt ruft er die `clean` Methode auf, mit der zur Referenz gehörenden Objekt ID. Die `clean` Methode entfernt den Eintrag in der Referenzliste zu *vmid* . Dabei wird auch die Squenznummer benötigt. Falls der Parameter *strong* `true` ist, dann ist der `clean` Aufruf das Ergebnis eines unerfolgreichen `dirty` Aufrufs. In diesem Fall muss die Sequenznummer für den Client, identifiziert mit Hilfe der *vmid*, beibehalten werden.

1.10.2. Die Lease Klasse

Ein Leasing enthält eine eindeutige Virtuelle Maschine, *vmid*, und eine Leasingdauer. Ein Lease Objekt wird benutzt, um das Leasing eines remote Objekts zu ermöglichen.

```
package java.rmi.dgc;
public final class Lease implements java.io.Serializable {
     public Lease(VMID id, long duration);
     public VMID getVMID();
     public long getValue();
}
```

Der `Lease` Konstruktor kreiert ein Leasingobjekt mit einer spezifischen VMID und Leasingdauer. Die VMID kann auch `null` sein. Die `getVMID` Methode liefert die Client VMID zum Leasingkontrakt. Die `getValue` Method liefert die Leasingdauer.

1.10.3. Die ObjID Klasse

Die Klasse `ObjID` wird benutzt, um remote Objekte eindeutig in einer virtuellen Maschine zu identifizieren. Jede Identifikation enthält eine Objektnummer und eine AdressraumID, welche pro spezifischer Host eindeutig ist.

Dem Objekt wird beim Exportieren eine ObjektID zugeordnet. Eine `ObjID` besteht aus einer Onjektnummer vom Typ `long` und einer eindeutigen Identifikation, für den Adressraum, einen UID.

```
package java.rmi.server;
public final class ObjID implements java.io.Serializable {
     public ObjID ();
     public ObjID (int num);
     public void write(ObjectOutput out) throws
                              java.io.IOException;
     public static ObjID read(ObjectInput in)
                              throws java.io.IOException;
     public int hashCode();
     public boolean equals(Object obj);
     public String toString();
}
```

Die erste Form des `ObjID` Konstruktors generiert eine eindeutige Objektidentität.
Der zweite Konstruktor generiert *well-known* ObjektIDs (wie jene, welche von der Registry und dem Distributed Garbage Collector) und verwendet als Argument eine dieser Nummern.

Eine ObjektID des zweiten Konstruktors wird nie imWiederspruch zu einer ObjektID des ersten Konstruktors stehen. Um dies sicherzustellen, wird die Objektnummer des Objekts `ObjID` gleich der“well-known” Zahl (Argument im Konstruktor) gesetzt und alle `UID` Felder werden auf 0 gesetzt.

Die `write` Methode verschlüsselt (marshalled) die Darstellung des ObID Objekts in einen Ausgabestrom.

Die `read` Methode konstruiert ein ObjID Objekt mit Hilfe des Inhalts des InputStreams.

Die `hashCode` Methode liefert eine Objektnummer als Hashcode.

Die `equals` Methode liefert true falls *obj* die selbe `ObjID` ist, inhaltsmässig.

Die `toString` Methode liefert eine Zeichenkettendarstellung des ObjektID Objekts.

Die Identifikation des Adressraumes wird nur dann in der Zeichenkette eingebaut, falls die ObjektID zu einem Objekt aus einem nicht lokalen Adressraum gehört

1.10.4. Die UID Klasse

Die Klasse UID ist eine Abstraktion zum Kreieren eindeutiger Identfier pro Host, auf dem er generiert wird.

Ein `UID` wird in einer Objekt ID `ObjID` zur Identifikation eines Adressraumes eingesetzt. Ein UID besteht aus einer auf diesem Host *host* (vom Typ `int`) eindeutigen Nummer, zu einem eindeutigen Zeitpunkt time (vom Typ `long`) und einem Zähler count (vom Typ `short`).

```
package java.rmi.server;
public final class UID implements java.io.Serializable {
     public UID();
     public UID(short num);
     public int hashCode();
     public boolean equals(Object obj);
     public String toString();
     public void write(DataOutput out) throws
                                        java.io.IOException;
     public static UID read(DataInput in) throws
                                        java.io.IOException;
}
```

Die erste Form des Konstruktors generiert eine Identifikationsnummer, die auf diesem Host eindeutig ist. Diese `UID` ist unter folgenden Bedingungen eindeutig:
a) die Maschine benötigt mehr als eine Sekunde um neu zu booten und
b) die Uhr der Maschine wird nie zurück gesetzt.

Um eine UID zu konstruieren, welche global eindeutig ist, wird das Paar
< `UID,InetAddress`> verwendet.

Die zweite Form des Konstruktors kreiert eine " *well-known"* `UID`. es gibt $2^{16}-1$ solcher "well-known" IDs. Eine so generierte ID wird nicht in Konflikt zu einer UID stehen, die mit dem Defaultkonstruktor generiert wurde.

Die Methoden `hashCode`, `equals`, und `toString` sind auch für UIDs definiert.

Zwei UIDs werden als gleich angesehen, falls sie inhaltsgleich sind.

Die Methode `write` schreibt die UID inden Ausgabestrom.
Die Methode `read` konstruiert eine UID aus dem Inhalt eines EIngabestroms.

1.10.5. Die VMID Klasse

Die Klasse `VMID` generiert universell eindeutige Identifier, unabhängig von der VM. Ein `VMID` Objekt enthält eine `UID` und eine Host Addresse. Ein `VMID` Objekt kann benutzt werden, um Client VMs zu identifizieren.

```
package java.rmi.dgc;
public final class VMID implements java.io.Serializable {
     public VMID();
     public static boolean isUnique();
     public int hashCode();
     public boolean equals(Object obj);
     public String toString();
}
```

Der `VMID` Default Konstruktor kreiert eine global eindeutige Identifikation, global im Sinne: unabhängig von der konkreten VM, unter folgenden Bedingungen:

- die Bedingungen für die Eindeutigkeit von Objekten der Klasse `java.rmi.server.UID` werden erfüllt und
- für den eindeutigen Host und die Lebensdauer des `UID` Objekts kann ein Adressraum gefunden werden

Ein `VMID` Objekt enthält die Hostadresse des Hosts, auf dem es kreiert wurde.
Aus Sicherheitseinschränkungen ist es zum Teil nicht möglich, die wahre Hostadresse zu erhalten.

Mit der Methode `isUnique` kann bestimmt werden, ob die in der lokalen VM generierten `VMIDs` universell eindeutig ist. Die Methode `isUnique` liefert true falls ein gültiger Hostname bestimmt werden kann; sonst wird false zurück gegeben.

Die Methoden `hashCode`, `equals` und `toString` sind auch für `VMID`s definiert. zwei VMIDs sind gleich, falls ihr Inhalt gleich ist.

1.11. RMI - Wire Protocol

1.11.1. Übersicht

Das RMI Protokoll verwendet zwei weitere Protokolle für die Kommunikation : Java Objekt Serialisierung und HTTP. Das Objekt Serialisierungsprotokoll wird zum Marshalling benutzt (verschlüsseln der Daten, schreiben der Daten in einen Outsgabestrom, lesen der Daten aus dem Eingabestrom, entschlüsseln der Daten / Rekonstruktion der Objekte).

Vom HTTP Protokoll wird die POST Methode verwendet. Die Rückgabe wird in den Body Teil geschrieben und muss anschliessend vom RMI Socket entschlüsselt werden.

Beide Protokolle verwenden eine unterschiedliche Syntax; beide Protokolle sind schriftlich festgehalten.
.

1.11.2. Grammatik Notation

Es gibt unterschiedliche Notationen zum Festhalten von Grammatiken. Die ältesten und bekanntesten sind die Backus Naur Form und die Syntax Grafen. Neue Formen basieren zum Beispiel auf sogenannten "Attributierten Grammatiken", wobei die Attribute sich auf die Sematik, die Bedeutung der einzelnen Konstrukte, beziehen.

- die Notation leht sich an jene der Java Sprachdefinition an, da beide Definitionen von Sun stammen.
- Kontrollcode wird mit Hilfe von Literalen beschrieben (in hexadezimaler Schreibweise)
- nichtterminale Symbole der Grammatik repräsentieren applikationsspezifische Werte der Methodenaufrufe.

Wir werden die Notation noch genauer kennen lernen, wenn wir sie brauchen.

1.11.3. RMI Transport Protokoll

Das RMI Transportprotokoll wird mit Hilfe eines *Stromes* dargestellt. Die Terminologie bezieht sich auf die Sicht des Clients. *Out* bezieht sich auf ausgehende Methodenaufrufe (an den Server), *In* bezieht sich auf ankommende Meldungen.

Der Transportheader wird nicht mit den Objektserialisierungsalgorithmen formattiert.

Stream:
Out
In

Eingabe und Ausgabe Streams treten in RMI paarweise auf (zu jedem Eingabestrem gehört ein Ausgabestream).

Der Ausgabestrom Out der Grammatik wird auf den Ausgabestrom eines Sockets (aus Sicht des Clients) abgebildet.

Der Eingabestrom In (der Grammatik) entspricht dem Eingabestrom des Sockets.

Da Eingabe und Ausgabe Ströme zusammen gehören, enthält der Header lediglich Acknowledge Information (wurde das Protokoll verstanden), da die sonst üblich Headerinformation (Versionsnummer, "magic number") sich aus dem Zusammenhang (Eingabestrom zu Ausgabestrom und umgekehrt) ergeben (s.u.).

1.11.3.1. Format eines Ausgabestromes

Der RMI Ausgabestrom besteht aus Transport *Header* Information plus einer Sequenz von *Meldungen*.

Alternativ kann ein Ausgabestrom Aufrufinformation für eine Methode in HTTP Protokollinformation enthalten.

Out:
    Header Messages
    HttpMessage
Header:
    `0x4a 0x52 0x4d 0x49` *Version Protocol*
Version:
    `0x00 0x01`
Protocol:
    StreamProtocol
    SingleOpProtocol
    MultiplexProtocol
StreamProtocol:
    `0x4b`
SingleOpProtocol:
    `0x4c`
MultiplexProtocol:
    `0x4d`
Messages:
    Message
    Messages Message

Messages werden entsprechend den Protokollvorgaben gewrapped, also mit Header und Abschluss- Informationen versehen.

Im Falle des *SingleOpProtocol* kann es sein, dass nach dem Header lediglich eine Meldung folgt *Message*, welche nicht weiter gewrapped / eingepackt ist.

Das *SingleOpProtocol* wird eingesetzt, falls der Aufruf ein einen HTTP Request eingebettet wird. In diesem Fall kann keine Kommunikation zwischen Anfrage und Antwort stattfinden (HTTP gestattet dies nicht, es handelt sich um ein "gedächtnisloses" Protokoll).

Im Falle des *StreamProtocol* und des *MultiplexProtocol* muss der Server mit einem Byte `0x4e` Acknowledment Support für das Protokoll antworten plus ein *EndpointIdentifier*, der Host Name und Port Nummer, die (aus Sicht des Servers) vom Client benutzt werden.

Damit kann der Client die Identität des Servers bestimmen, falls er andersweitig dazu nicht in der Lage ist, zum Beispiel aus Sicherheitsgründen.

Der Client antwortet seinerseits mit einem andern *EndpointIdentifier*, der jetzt aber die Informationen über den Client enthalten: seinen Default Endpoint für eintreffende Verbindungen.

Der Server kann damit im Falle des *MultiplexProtocol* den Client identifizieren.

Im Falle des *StreamProtoco*l werden, nachdem die Endpoint Informationen ausgetauscht sind, die *Messages* ohne weitere Wrapper über den Output Stream (des Clients) versandt.

Im Falle des *MultiplexProtoco*l wird eine Socket Verbindung verwendet Virtuelle Verbindungen, welche über eine solche multiplexte Verbindung aufgebaut wird, besteht aus einer Serie *Messages* wie unten beschrieben.

Es gibt drei Arten ausgehenden Messages *Cal*l, *Ping* und *DgcAck:*

- ein *Call* verschlüsselt einen Methodenaufruf.
- ein *Ping* ist eine Meldung auf Transportlevel. Damit kann man testen, ob die Verbindung zur remote VM noch besteh
- ein *DGCAck* ist eine Bestätigung an den Distributed Garbage Collector des Servers. Damit wird bestätigt, dass ein remote Objekt vom Server durch den Client empfängen wurde.

.

Message:

Call
Ping
DgcAck

Call:

`0x50` *CallData*

Ping:

`0x52`

DgcAck:

`0x54` *UniqueIdentifier*

1.11.3.2. Format eines Input Stream

Man unterscheidet drei Arten von Input Messages :

1. *ReturnData*,
2. *HttpReturn* und
3. *PingAc*k.

ReturnData ist das Ergebnis eines normalen RMI Aufrufs.

Ein *HttpReturn* ist das Ergebnis eines remote Methodenaufrufs mit Hilfe des HTTP Protokolls.

Ein *PingAck* ist die Bestätigung einer *Ping* Message.

In:

ProtocolAck Returns
ProtocolNotSupported
HttpReturn

ProtocolAck:

`0x4e`

ProtocolNotSupported:

`0x4f`

Returns:
Return
Returns Return

Return:
ReturnData
PingAck

ReturnData:
`0x51` *ReturnValue*

PingAck:
`0x53`

1.11.4. RMI's Einsatz des Object Serialization Protokolls

Aufrufdaten und Rückgabewerte in RMI Aufrufen benutzen das Java Objekt Serialisierungsprotokoll. *CallData* jedes Methodenaufrufs wird repräsentiert

- durch den *ObjectIdentifier* (das Target des Aufrufs),
- eine *Operation* (eine Zahl, welche die Methode charakterisiert),
- einen *Hash* (eine Zahl, mit deren Hilfe überprüft wird, dass Client Stub und Server Skeleton das selbe Stub Protokoll benutzen),
- gefolgt von einer Liste von Nullen oder weiteren *Arguments* des Aufrufs.

Im JDK1.1 Stub Protokoll war die *Operation* die Methodennummer, welche vom `rmic` der Methode zugeordnet wurde und *Hash* war der Stub / Skeleton Hash, der gliech dem Interface Hash des Stub ist.

Im JDK1.2 Stub Protokoll (also den Stubs, welche mit der `rmic` -v1.2 generiert wurden) ist *Operation* -1 und der *Hash* ist ein Hash der die aufzurufende Methode darstelt.

Syntax:

CallData:
ObjectIdentifier Operation Hash Arguments$_{opt}$
ObjectIdentifier:
ObjectNumber UniqueIdentifier
UniqueIdentifier:
Number Time Count
Arguments:
Value
Arguments Value
Value:
Object
Primitive

Ein *ReturnValue* eines RMI Aufrufs besteht aus einem Return Code, mit dem die erfolgreiche oder fehlerhafte Beendigung des Aufrufs angezeigt wird, einem *UniqueIdentifier,*mit dem die Rückgabe eindeutig identifiziert werden kann (dieser wird auch verwendet, um eine DGCAck zu senden, falls nötig) gefolgt vom Ergebnis: entweder dem Rückgabewert *Value*

oder der Ausnahme *Exception*, falls eine gworfen wurde

.

ReturnValue:

`0x01` *UniqueIdentifier Value*$_{eopt}$

`0x02` *UniqueIdentifier Exception*

Bemerkung – *ObjectIdentifie*r, *UniqueIdentifier,* und *EndpointIdentifier* verwenden eigene `write` Methoden, die sich von der `writeObject` Methode der Object Serialization unterscheiden);
die `write` Methode für den jeweiligen Datentyp fügt jeweils die netsprcehnden Komponentendaten dem Output Stream hinzu.

1.11.4.1. Class Annotation und Class Loading

RMI überschreibt die Methoden `annotateClass` und `resolveClass` von `ObjectOutputStream` und `ObjectInputStream`. Jede Klasse wird durch deb Codebase Pfad, bzw. URL ergänzt (Lokation der Classdatei). In der `annotateClass` Methode, wird der Classloader, der die Klassen laden muss, abgefragt nach der Codebase URL. Falls der Classloader existiert und eine definierte Codebase besitzt, dann wird diese in den Ausgabestrom geschrieben, mit Hilfe der `ObjectOutputStream.writeObject` Method; sonst wird ein null Objekt in den Objektstrom gestellt, mit Hilfe der `writeObject` Methode.

Bemerkung: die Klassen der Java Packages,"java...", werden aus Optimierungsgründen nicht in den Ausgabestrom gestellt, da diese Klassen jederzeit zur Verfügung stehen.

Die Klassen Annotation wird bei der Deserialisierung mit Hilfe der Methode `ObjectInputStream.resolveClass` aufgelöst. Die `resolveClass` Methode liest zuerst die Annotation mit Hilfe der `ObjectInputStream.readObject` Methode. Falls die Annotation, die URL der Codebase, nicht null ist, dann wird versucht die Klasse zu laden. Dies geschieht mit Hilfe einer `java.net.URLConnection,` um die Classbytes zu lesen. Dies ist der selbe Mechanismus, wie beim Classloader der Applets in einem Browser.

1.11.5. RMI's Einsatz des HTTP POST Protokolls

Methodenaufrufe durch eine Firewall sind mit Hilfe des HTTP Protokolls möglich, konkret mit Hilfe der POST Methode. Die URL für einen solchen Einsatz sieht folgendermassen aus:

1. http://<host>:<port>/
2. http://<host>:80/cgi-bin/java-rmi?forward=<port>

Die erste Form der Adressierung spezifiziert den Host *host* und den Port *port* des RMI Servers.

Die zweite Form verwendet das cgi Skript, welches wir bereits kennen gelernt haben und welches im Anhang nochmals aufgelistet wird.

Ein *HttpPostHeader* ist ein Standard HTTP Header für eine POST Anforderung.
Ein *HttpResponseHeader* ist eine Standard HTTP Anwort auf einen POST Request.

Falls der Statuscode der Antwort nicht 200 ist, wird angenommen, dass keine Antwort *Return* geliefert wurde.

Pro HTTP Post Request wird lediglich ein RMI Aufruf übermittelt.

Syntax:

HttpMessage:
HttpPostHeader Header Message
HttpReturn:
HttpResponseHeader Return

Bemerkung – Im Header *Header* einer *HttpMessage* erscheint nur das *SingleOpProtocol. HttpReturn* enthält kein Prootkoll Acknowledgment Byte.

1.11.6. Applikation spezifische Werte für RMI

Die folgende Tabelle listet die nichtterminalen Symbole, welche applikationsspezifische Werte darstellen, die von RMI benutzt werden.

Die Tabelle zeigt die Verbindung der Symbolde zu den entsprechenden Datentypen.
Jeder Datentyp wird mit Hilfe des Protokolls, in das er eingebettet ist, formattiert.

Count	`short`
Exception	`java.lang.Exception`
Hash	`long`
Hostname	`UTF`
Number	`int`
Object	`java.lang.Object`
ObjectNumber	`long`
Operation	`int`
PortNumber	`int`
Primitive	`byte, int, short, long...`
Time	`long`

1.11.7. RMI's Multiplexing Protokoll

Multiplexing wird dann wichtig, wenn zwei Endpoints jeder mehrere Full-Duplex Verbindungen zum andern Endpoint öffnen kann, in einer Umgebung, in der lediglich einer der Endpoints eine solche bidirektionale Verbindung mit Hilfe eines beliebigen Mechansmus (zum Beispiel TCP Connection) *eröffnen* kann.

RMI verwendet diese einfache Multiplexing Protokoll, um einem Client die Möglichkeit zu geben, eine Verbindung zu einem RMI Server aufzubauen, in Situationen, in denen dieser Verbindungsaufbau sonst nicht möglich wäre.

Beispielanwendung wäre ein Applet mit einem SecurityManager, der die Kreation eines ServerSockets für den Emfang eintreffender Verbindungsanforderungen verbietet. Damit ist

es auch nicht möglich, RMI Objekte zu exportieren und remote Serviceanfragen für eintreffende Anfragen auszuführen.
Falls das Applet eine normale Socketverbindung zum Codebase Host aufbauen kann, dann kann mit Hilfe des Multiplexing Protokolls über diese Verbindung eine Methode eines MI Objekts aufgerufen werden.

1.11.7.1. Definitionen

Dieser Abschnitt definiert einige Begriffe, die bei der Beschreibung des Protokolls benötigt werden.

Ein *endpoint* ist einer der beiden Benutzer einer Verbindung mit Hilfe des Multiplexing Protokolls.

Das Multiplexing Protokoll wird oberhalb einer existierenden bidirektionalen Verbindung aufgabaut. Diese Verbindung wird in der Regel durch einen der Endpoints aufgebaut. Die gegenwärtige Version von RMI setzt voraus, dass diese Verbindung eine Socket Verbindung ist, realisiert mit Hilfe von `java.net.Socket`. Diese Verbindung bezeichnet man als die konkrete Verbindung, *concrete connection.*

Das multiplexing Protokoll ermöglicht denEinsatz von *virtuellen Connection*s
Alle virtuellen Verbindungen zusammen bezeichnet man als die *multiplexed Connection.*

Virtuelle Verbindungen können jeweils von beiden Enden der Kommunikation gestoppt, unterbrochen werden.

Eine virtuelle Verbindung wird mit Hilfe einer 16 bit Integer Zahl, dem *connection identifier*, identifiziert. Es gibt somit maximal 65,536 mögliche virtuelle Verbindungen in einer multiplexten Verbindung. Je nach Implementation sind mehr oder weniger als virtuelle Verbindungen möglich (maximal 65'536).

1.11.7.2. Verbindungszustand und Flow Control

Das Multiplexing Protokoll definiert einige Operationen: OPEN, CLOSE, CLOSEACK, REQUEST und TRANSMIT.

Das exakte Format wird weiter unten spezifiziert.

Die Operationen OPEN, CLOSE und CLOSEACK kontrollieren, ob Verbindungen geöffnet oder geschkossen werden.
REQUEST und TRANSMIT Operationen werden zum Senden der Daten über eine offene Verbindung eingesetzt.

Connection States : Zustände einer Verbindung
Eine virtulelle Verbindung ist offen bezogen auf einen bestimmten Endpunkt, falls der Endpunkt eine OPEN Operation gesendet oder empfangen hat.

Eine virtuelle Verbindung ist *pending close* bezogen auf einen bestimmten Endpunkt, falls der Endpunkt eine CLOSE Operation verlangt hat, aber die CLOSE oder CLOSEACK Operation für diese Connection noch nicht durchgeführt wurden.
.
Eine virtuelle Verbindung ist geschlossen, *closed,* in Bezug auf einen bestimmten Endpunkt, falls die Verbindung nie geöffnet wurde oder ein CLOSE oder eine CLOSEACK Operation für diese Verbindung abgesetzt wurden,

Flow Control
Das multiplexing Protokoll verwendet einen einfachen Packeting Flow Control Mechanismus, um mehrere virtuelle Verbindungen parallel über eine Verbindung zu erlauben.

Jeder Endpunkt besitzt zwei Zustandswerte pro Verbindung :
- wie viele Bytes wurden verlangt, aber noch nicht übermittelt (*Input Request count)*
- wie viele Bytes wurden vom andern Endpunkt verlangt aber nicht geliefert (*Output Request Count*)

Diese Zähler werden erhöht, falls eine REQUEST Operation empfangen wird; er wird reduziert, falls eine TRANSMIT Operation gesendet wurde.

Falls einer der Werte negativ wird, wird eine "protocol violation" gesetzt.

1.11.7.3. Protokoll Format

Der Bytestrom des multiplexing Protokolls besteht aus einem Strom von Datensätzen variabler Länge. Als erstes Byte wird ein Operationscode übermittelt:

Wert	**Name**
0xE1	OPEN
0xE2	CLOSE
0xE3	CLOSEACK
0xE4	REQUEST
0xE5	TRANSMIT

Falls das erste Byte nicht aus der obigen Liste ist, tritt eine "protocol violation" auf.

1.11.7.3.1. OPEN Operation

Grösse (Bytes)	**Name**	**Beschreibung**
1	*opcode*	Operationscode (OPEN)
2	*ID*	Connection Identifier

Ein Endpoint sendet eine OPEN Operation, um die bezeichnete Verbindung zu öffnen. Es ist eine "protocol violation" falls *ID* eine Verbindung bezeichnet, die bereits offen oder pending ist. Nachdem die Verbindung geöffnet wurde, werden die Zähler für input und request count states auf 0 gesetzt.

1.11.7.3.2. CLOSE Operation

Grösse	**Name**	**Beschreibung**
1	*opcode*	Operationcode (OPEN)
2	*ID*	Connection Identifier

Diese Operation schliesst die Verbindung, die mit Hilfe der ID identifiziert wird.

1.11.7.3.3. CLOSEACK Operation

Grösse	Name	Beschreibung
1	*opcode*	Operationscode (OPEN)
2	*ID*	Connection Identifier

Damit wird der CLOSE Request eines Endpunkts bestätigt.

1.11.7.3.4. REQUEST Operation

Grösse	Name	Beschreibung
1	*opcode*	Operationscode (OPEN)
2	*ID*	Connection Identifier
4	*count*	Anzahl Bytes, die noch gesendet werden müssen

.

Der Wert von *count* ist eine 32 bit Integer Zahl mit Vorzeichen.

1.11.7.3.5. TRANSMIT Operation

Grösse (Bytes)	Name	Beschreibung
1	*opcode*	Operationscode (OPEN)
2	*ID*	Connection Identifier
4	*count*	Anzahl Bytes, die übermittelt werden

1.11.7.3.6. Protocol Violations

Protokoll Violations führen zum sofortigen Abbruch der multiplexten Verbindung.

1.11.8. Anhang : Das java-rmi Perl Skript für HTTP POST

```
java   \
  -DAUTH_TYPE=$AUTH_TYPE \
  -DCONTENT_LENGTH=$CONTENT_LENGTH \
  -DCONTENT_TYPE=$CONTENT_TYPE \
  -DDOCUMENT_ROOT=$DOCUMENT_ROOT \
  -DGATEWAY_INTERFACE=$GATEWAY_INTERFACE \
  -DHTTP_ACCEPT="$HTTP_ACCEPT" \
  -DHTTP_CONNECTION=$HTTP_CONNECTION \
  -DHTTP_HOST=$HTTP_HOST \
  -DHTTP_USER_AGENT="$HTTP_USER_AGENT" \
  -DPATH_INFO=$PATH_INFO \
  -DPATH_TRANSLATED=$PATH_TRANSLATED \
  -DQUERY_STRING=$QUERY_STRING \
  -DREMOTE_ADDR=$REMOTE_ADDR \
  -DREMOTE_HOST=$REMOTE_HOST \
  -DREMOTE_IDENT=$REMOTE_IDENT \
  -DREMOTE_USER=$REMOTE_USER \
  -DREQUEST_METHOD=$REQUEST_METHOD \
  -DSCRIPT_NAME=$SCRIPT_NAME \
  -DSERVER_NAME=$SERVER_NAME \
  -DSERVER_PORT=$SERVER_PORT \
  -DSERVER_PROTOCOL=$SERVER_PROTOCOL \
  -DSERVER_SOFTWARE=$SERVER_SOFTWARE \
  sun.rmi.transport.proxy.CGIHandler
```

1.12. RMI - Exceptions und Properties

Dies ist der Anhang zum RMI Skript!

1.12.1. Exceptions

Die folgenden Ausnahmen sind in den verschiedenen Klassen definiert worden und hier einfach der besseren Übersicht halber nchmals zusammen gestellt.

1.12.1.1. Exceptions während des Remote Objekt Exports

Falls ein remote Objekt kreiert wird, welches das `UnicastRemoteObject` erweitert, wird das Objekt exportiert, das heisst: es kann Anfragen von andern Java VMs empfangen, die mit Hilfe von RMI kommunizieren.

Falls das Objekt nicht `UnicastRemoteObject` erweitert, dann muss die `java.rmi.server.UnicastRemoteObject.exportObject` Methode aufgerufen werden, um das Objekt zu exportieren:

Exception	**Context**
`java.rmi.StubNotFoundException`	1. Class of stub not found. 2. Name collision with class of same name as stub causes one of these errors: • Stub can't be instantiated. • Stub not of correct class. 3. Bad URL due to wrong codebase. 4. Stub not of correct class.

`java.rmi.server.`
`SkeletonNotFoundException`
in JDK1.2 verworfen
	1. Class of skeleton not found.
	2. Name collision with class of same name as skeleton causes one of these errors:
	• Skeleton can't be instantiated.
	• Skeleton not of correct class.
	3. Bad URL due to wrong codebase.
	4. Skeleton not of correct class.

java.rmi.server.ExportException	The port is in use by another VM.

1.12.1.2. Exceptions während RMI Calls

Exception	**Context**
`java.rmi.UnknownHostException`	Unknown host
`java.rmi.ConnectException`	Connection refused to host
`java.rmi.ConnectIOException`	I/O error creating connection
`java.rmi.MarshalException`	I/O error marshaling transport header, marshaling call header, or marshaling arguments.
`java.rmi.NoSuchObjectException`	Attempt to invoke a method on an object that is no longer available.
`java.rmi.StubNotFoundException`	Remote object not exported.
`java.rmi.activation.` `ActivateFailedException`	Thrown by RMI runtime when activation fails during a remote call to an activatable object

1.12.1.3. Exceptions oder Errors bei der Rückgabe

Exception	Context
`java.rmi.UnmarshalException`	1. Corrupted stream leads to either an I/O or protocol error when: • Marshaling return header. • Checking return type. • Checking return code. • Unmarshaling return. 2. Return value class not found.
`java.rmi.UnexpectedException`	An exception not mentioned in the method signature occurred (excluding runtime exceptions). The UnexpectedException exception object contains the underlying exception that was thrown by the server

1.12.1.4. Mögliche Ursachen für java.rmi.ServerException

Es gibt mehrere Unsachen für Exceptions auf dem Server. Diese Exceptions werden in `java.rmi.ServerException`n zusammen gefasst.
Das heisst: `java.rmi.ServerException` enthält die Exception Informationen.

Exception	Context
`java.rmi.server.SkeletonMismatchException` *note: this exception is deprecated in JDK1.2*	Hash mismatch of stub and skeleton
`java.rmi.UnmarshalException`	I/O error unmarshaling call header. I/O error unmarshaling arguments.
`java.rmi.MarshalException`	Protocol error marshaling return.
`java.rmi.RemoteException`	Method number out of range due to corrupted stream.

1.12.1.5. Naming Exceptions

Die folgende Liste bezieht sich auf die Registry und die Naming Klasse.

Exception	Context
`java.rmi.AccessException`	Operation disallowed. The registry restricts bind, rebind, and unbind to the same host. The lookup operation can originate from any host.
`java.rmi.AlreadyBoundException`	Attempt to bind a name that is already bound.
`java.rmi.NotBoundException`	Attempt to look up a name that is not bound.
`java.rmi.UnknownHostException`	Attempt to contact a registry on an unknown host.

1.12.1.6. Activation Exceptions

Die folgende Liste enthält Exceptions aus dem `java.rmi.activation` API.

Exception	Context
`java.rmi.activation.ActivateFailedException`	Thrown by RMI runtime when activation fails during a remote call to an activatable object.
`java.rmi.activation.ActivationException`	General exception class used by the activation interfaces and classes.
`java.rmi.activation.UnknownGroupException`	Thrown by methods of the activation classes and interfaces when the ActivationGroupID parameter or ActivationGroupID in an ActivationGroupDesc parameter is invalid.
`java.rmi.activation.UnknownObjectException`	Thrown by methods of the activation classes and interfaces when the ActivationID parameter is invalid.

1.12.1.7. Andere Exceptions

Exception	Context
`java.rmi.RMISecurityException` *note: this exception is deprecated in JDK1.2*	A security exception that is thrown by the RMISecurityManager.
`java.rmi.server.ServerCloneException`	Clone failed.
`java.rmi.server.ServerNotActiveException`	Attempt to get the client host via the RemoteServer.getClientHost method when the remote server is not executing in a remote method.
`java.rmi.server.SocketSecurityException`	Attempt to export object on an illegal port.

1.12.2. Properties in RMI

Properties kann man auf der Kommandozeile setzen oder aus Applets.
Beispiel eines Einsatzes wäre :

```
java -Djava.rmi.server.codebase=file://c:\test/ ...
```

1.12.2.1. Server Properties

Die folgende Tabelle enthält eine Liste der typischen Eigenschaften, die serverseitig einsetzbar sind. Typischerweise werden diese in Applets gesetzt.

Property	Beschreibung
`java.rmi.server.codebase`	Indicates the codebase URL of classes originating from the VM. The codebase property is used to annotate class descriptors of classes *originating* from a VM so that the class for an object sent as a parameter or return value in a remote method call can be

	loaded at the receiver.
`java.rmi.server.disableHttp`	If set to true, disables the use of HTTP for RMI calls. This means that RMI will never resort to using HTTP to invoke a call via a firewall. Defaults to false (HTTP usage is enabled).
`java.rmi.server.hostname`	RMI uses IP addresses to indicate the location of a server (embedded in a remote reference). If the use of a hostname is desired, this property is used to specify the fully-qualified hostname for RMI to use for remote objects exported to the local VM. The property can also be set to an IP address. Not set by default.
`java.rmi.dgc.leaseValue`	Sets the lease duration that the RMI runtime grants to clients referencing remote objects in the VM. Defaults to 10 minutes
`java.rmi.server.logCalls`	If set to true, server call logging is turned on and prints to stderr. Defaults to false.
`java.rmi.server.useCodebaseOnly`	If set to true, when RMI loads classes (if not available via CLASSPATH) they are only loaded using the URL specified by the property java.rmi.server.codebase.
`java.rmi.server.` `useLocalHostname`	useLocalHostname If the java.rmi.server.hostnameproperty is not set and this property is set, then RMI will not use an IP address to denote the location (embedded in remote references) of remote objects that are exported into the VM. Instead, RMI will use the value of the call to the method java.net.InetAddress.getLocalHost.

1.12.2.2. Activation Properties

Die folgende Liste enthält Aktivierungsparameter, die auf der Kommandozeile oder im Applet angegeben werden können.

Property	**Beschreibung**
`java.rmi.activation.port`	The port number on which the ActivationSystem is exported. This port number should be specified in a VM if the activation daemon **rmid** uses a port other than the default.
`java.rmi.activation.activator.class`	The class that implements the interface
`java.rmi.activation.Activator`	This property is used internally to locate the resident implementation of the Activator from which the stub class name can be found.

1.12.2.3. Andere Properties

Die folgenden Properties wurden alle in JDK 1.2 verworfen. Sie dienten der Lokalisation von Klassen innerhalb eines Implementations Packages.

Property	**Beschreibung**
`java.rmi.loader.packagePrefix (deprecated in JDK1.2)`	The package prefix for the class that implements the interface java.rmi.server.LoaderHandler. Defaults to sun.rmi.server.
`java.rmi.registry.packagePrefix (deprecated in JDK1.2)`	The package prefix for the class that implements the interface java.rmi.registry.RegistryHandler. Defaults to sun.rmi.registry.
`java.rmi.server.packagePrefix (deprecated in JDK1.2)`	The server package prefix. Assumes that the implementation of the server reference classes (such as UnicastRef and UnicastServerRef) are located in the package defined by the prefix. Defaults to sun.rmi.server.

JAVA REMOTE METHODE INVOCATION