



Netzprogrammierung Verteilte Objekte in Java RMI

Prof. Dr.-Ing. Robert Tolksdorf
Freie Universität Berlin
Institut für Informatik
Netzbasierte Informationssysteme
mailto: tolk@inf.fu-berlin.de
<http://www.robert-tolksdorf.de>



Überblick

1. Verteilte Objekte
2. Objektreferenzen
3. Parametersemantik
4. RMI Fehler
5. Code nachladen
6. Serialisierung



Verteilte Objekte

RMI vs. Sockets

- RMI
 - Höheres Abstraktionsniveau
 - Interaktionsform fest vorgegeben
 - Übermittlung getypter Daten
 - Klassenübermittlung
 - ...
- Sockets
 - Kleinster gemeinsamer Nenner des Internets
 - Nicht vorgegebene Interaktionsform
 - Übermittlung ungetypter Byteströme
 - Effizienter
 - ...

Verteilte Objekte

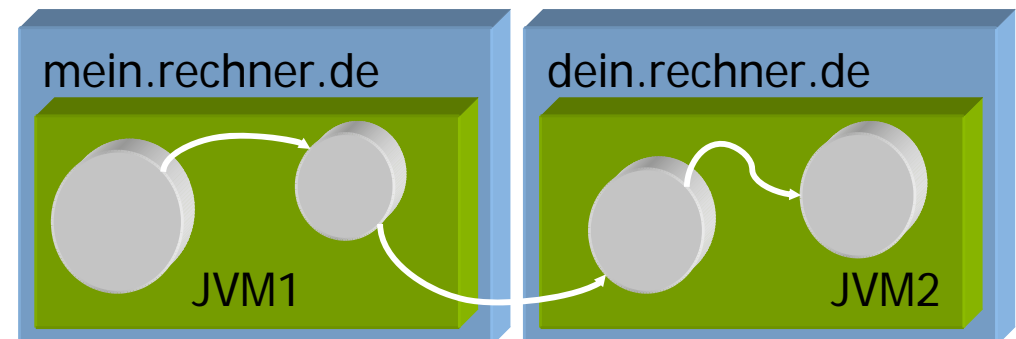
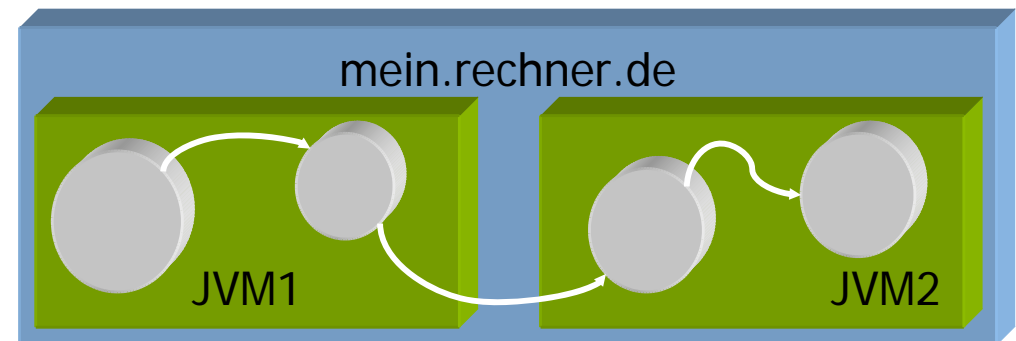
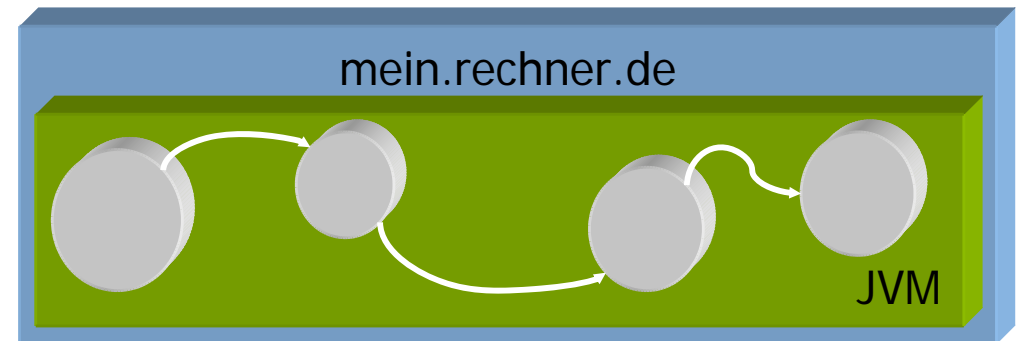
- RPC übertragen auf OO-Welt:
Entfernter Methodenaufruf
- Interaktionsmuster in OO-Sprachen:
 - Objekte tauschen Mitteilungen aus
 - Beim Empfang einer Mitteilung führt ein Objekt eine Methode aus und schickt eventuelle Ergebnisse
 - Modell sagt nichts über Verteilung aus
- Rollen der Partner
 - Aufrufer – Aufgerufener
 - Dienstnutzer – Dienstanforderer
 - Client – Server
- Verteilte Objekte in Java:
Remote Method Invocation, RMI

Lokale vs. verteilte Objekte

<i>Lokales Objektmodell</i>	<i>Verteiltes Objektmodell</i>
Aufruf an <i>Objekten</i>	Aufruf an <i>Interfaces</i>
Parameter und Ergebnisse als <i>Referenzen</i>	Parameter und Ergebnisse als <i>Kopien</i>
Alle Objekte fallen <i>zusammen</i> aus	<i>Einzelne</i> Objekte fallen aus
<i>Keine</i> Fehlersemantik	<i>Komplizierte</i> Fehlersemantik (Referenzintegrität, Netzfehler, Sicherheit etc.)
...	

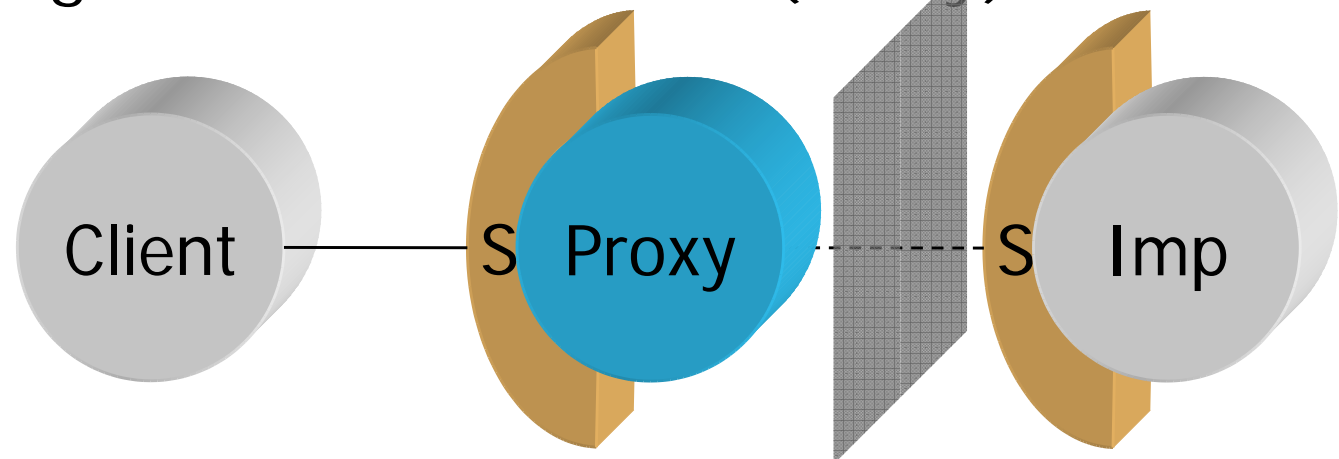
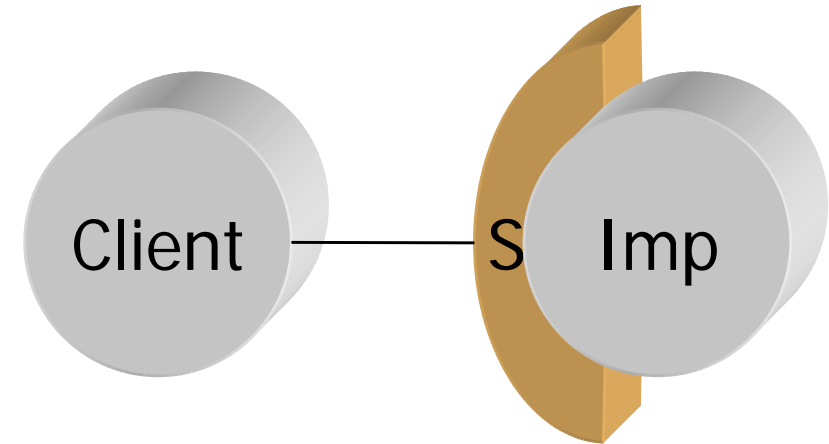
Lokale vs. verteilte Java-Programme

- Ein Java Programm arbeitet in einer virtuellen Java Maschine (JVM)
- Zwischen JVMs können mit *Remote Method Invocation* Methoden an Objekten aufgerufen werden
- JVMs können auf unterschiedlichen Internet-Rechnern laufen
- Sie müssen es aber nicht...



Schnittstellen

- Objektschnittstellen definieren Methoden des Objekts
- Unterschiedliche Implementierungen für gleiche Schnittstelle S
- Modulschnittstellen des RPC werden auf Objektschnittstellen abgebildet
- Aufrufweiterleitung durch Stellvertreter (Proxy)



Lokales Zählerobjekt

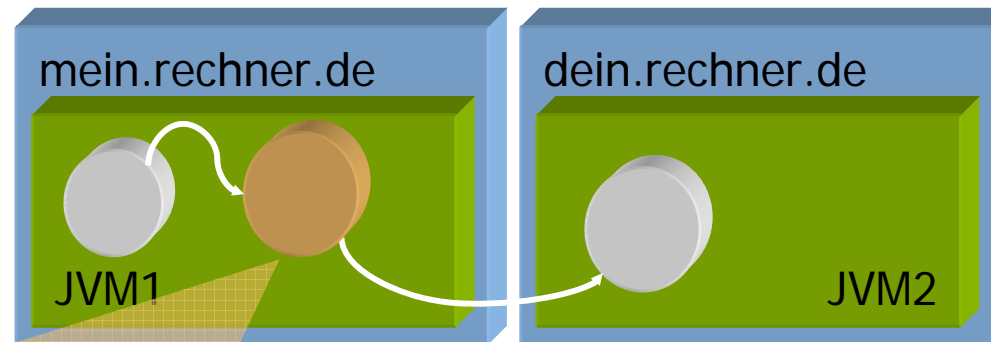
- Lokale nutzbares Objekt:

```
class LocalCounterImplementation {
    int counter;
    public LocalCounterImplementation() { }
    public void add(Integer i) {
        counter += i.intValue();
    }
    public Integer value() {
        return(new Integer(counter));
    }
}
```
- Könnte auch folgende Schnittstelle implementieren:

```
public interface LocalCounter {
    /* Addieren */
    public void add(Integer i);
    /* Abfragen */
    public Integer value();
}
```
- Aufruf: `c.add(new Integer(10));`

Entferntes Zählerobjekt

- Auf Aufruferseite wird mit einem Interface gesprochen:

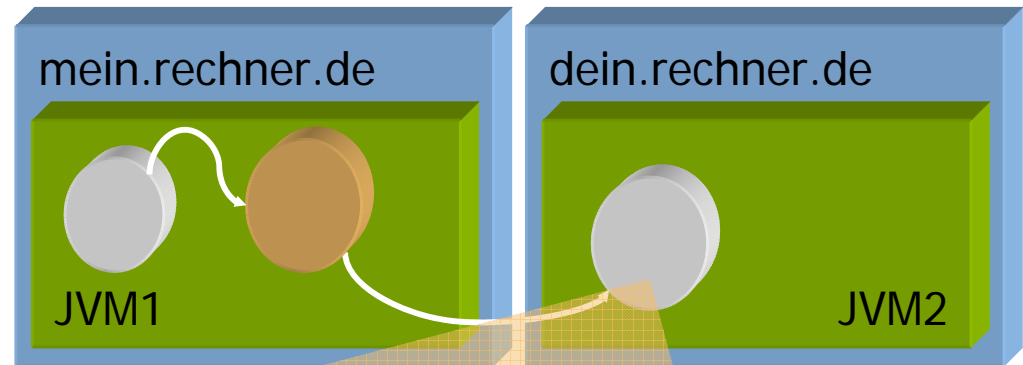


```
public interface Counter extends java.rmi.Remote {  
    /* Addieren */  
    public void add(Integer i) throws java.rmi.RemoteException;  
    /* Abfragen */  
    public Integer value() throws java.rmi.RemoteException;  
}
```

- Schnittstelle macht Fehlermöglichkeit explizit

RMI: Erbringerseite

- Serverobjekt implementiert das Interface:



```
import java.rmi.*;
public class CounterServer extends
    java.rmi.server.UnicastRemoteObject
    implements Counter {
    int counter;
    public CounterServer() throws java.rmi.RemoteException { }
    public void add(Integer i) throws java.rmi.RemoteException {
        counter += i.intValue();
    }
    public Integer value() throws java.rmi.RemoteException {
        return(new Integer(counter));
    }
}
```

Aufruf eines entfernten Objekts

- Aufruf durch Aufruf einer Methode an Interface
- Tatsächlich wird damit eine Methode am Proxy aufgerufen:

```
Counter c;
```

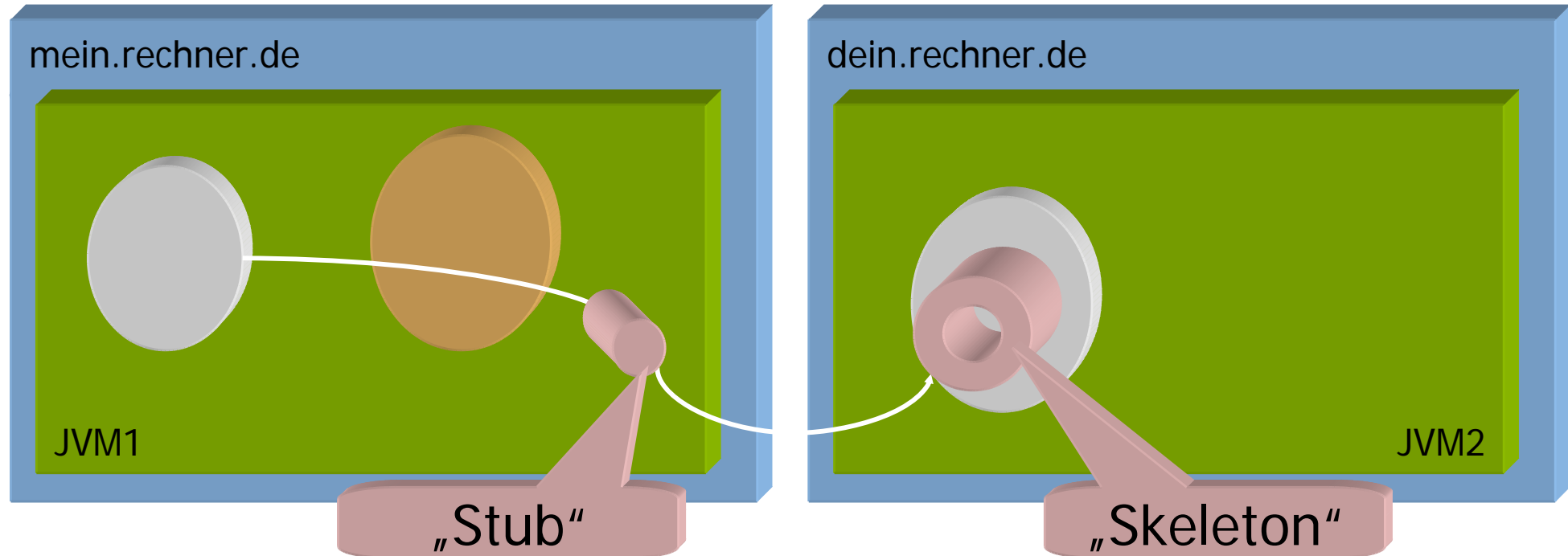
```
...
```

```
try {  
    c.add(new Integer(10));  
    System.out.println((c.value()).intValue());  
} catch (Exception e) {  
    System.err.println(e.getMessage());  
}
```

- *Unterschied: Fehlermöglichkeit*

rmic: Compiler für Verbindungsstücke

- „Verbindungsstücke“ werden automatisch erzeugt



- Aufruf `rmic ServerKlasse` erzeugt `ServerKlasse_stub.class` und `ServerKlasse_skel.class`

Compilieren und ausführen

- Compilieren:

```
>javac Counter.java  
>javac CounterServer.java  
>rmic CounterServer  
>javac CounterUser.java
```

- In 4 JVMs ausführen:

```
>rmiregistry    >java CounterServer
```

```
>java CounterUser
```

```
10
```

```
>java CounterUser
```

```
20
```



Objektreferenzen

Objektreferenzen

```
Counter c;
```

```
...
```

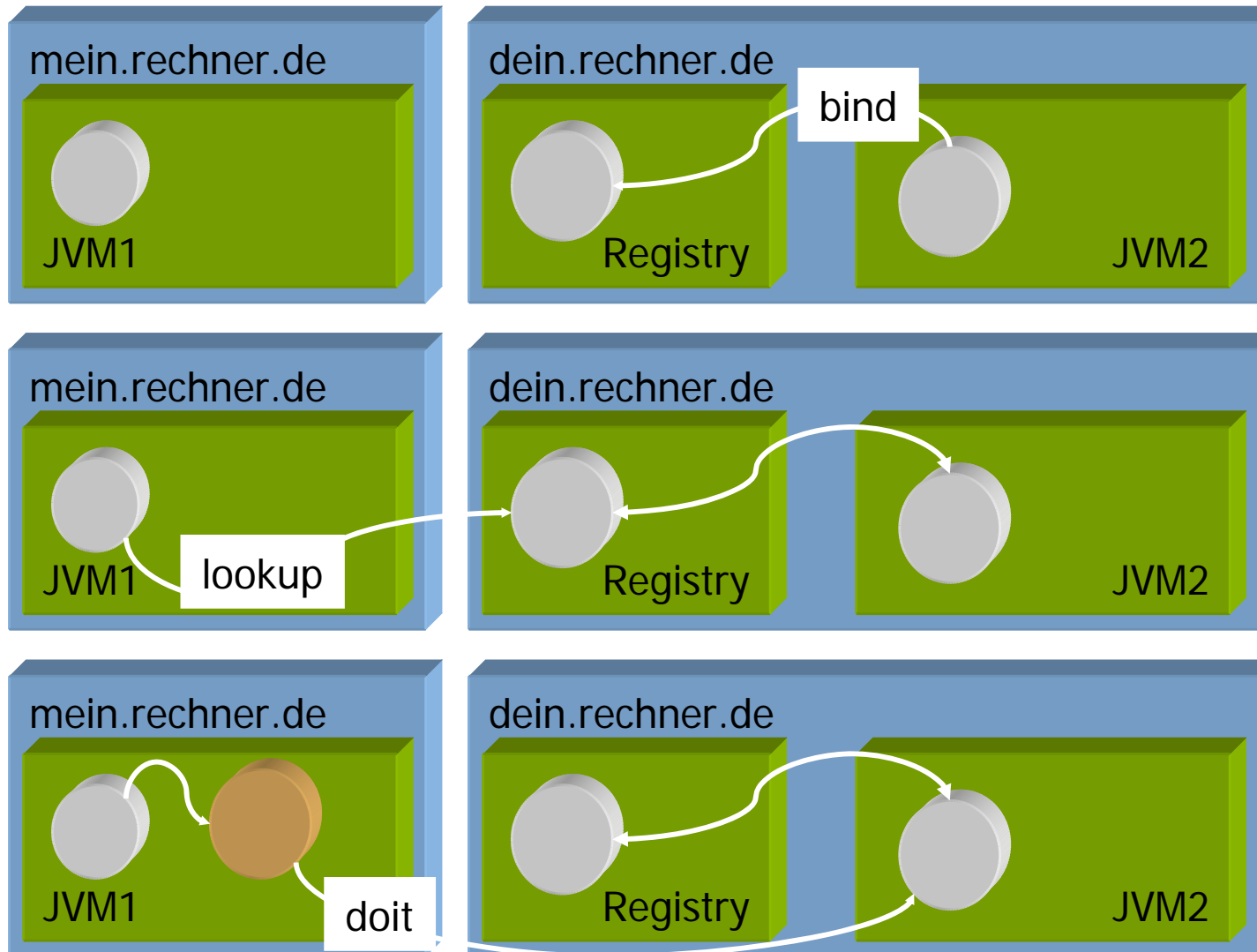
```
try {
```

```
    c.add(new Integer(10));
```

- Woher kennt der Aufrufer eigentlich das entfernte Objekt?
- Genauer: Wohin schickt der Proxy eigentlich den Methodenaufruf?
- RMIRegistry: „Verzeichnisdienst“ für Objekte

Referenzen auf entfernte Objekte

- *Registry* Objekt liefert Referenzen auf Objekte



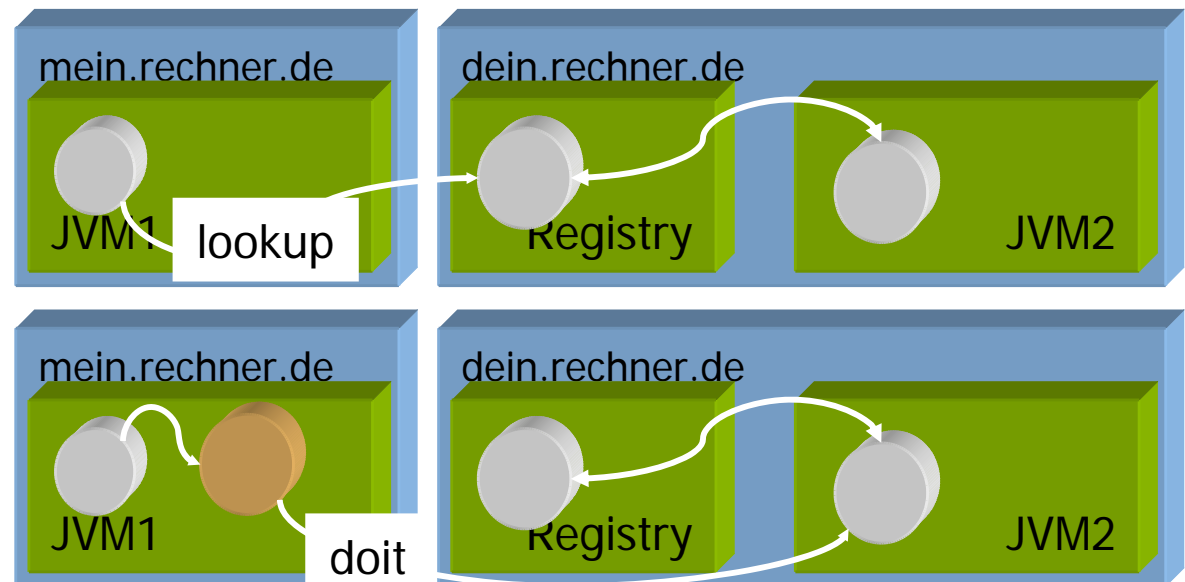
Beispiel: Server-Programm meldet sich an

```
public class CounterServer extends
    java.rmi.server.UnicastRemoteObject
    implements Counter {
    //...
    public static void main(String argv[]) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            CounterServer c = new CounterServer();
            // Naming.bind("rmi://dein.rechner.de/Teilnehmer",c);
            Naming.bind("rmi://" + (InetAddress.getLocalHost()).getHostName()
                + "/Teilnehmer",c);
        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }
}
```

- URL-Schema: *rmi://registryrechner:registryport/pfad*
- Entfernter Zugang zu Registry ist eingeschränkt

Client Programm erhält Referenz

```
import java.net.*;
public class CounterUser {
    public static void main(String argv[]) {
        try {
            Counter c = (Counter)
                java.rmi.Naming.lookup("rmi://dein.rechner.de/Teilnehmer");
            c.add(new Integer(10));
            System.out.println((c.value()).intValue());
        } catch (Exception e) {
            System.err.println(
                e.getMessage());
        }
    }
}
```



rmiregistry

- JDK Programm `rmiregistry` ist RMI-Objekt für Interface `java.rmi.registry.Registry`
- Methoden
 - `void bind(String name, Remote obj)`
`void rebind(String name, Remote obj)`
Binden eines Objektes unter einem Namen
 - `Remote lookup(String name)`
Referenz auf gebundenen Objekt erfragen
 - `void unbind(String name)`
Bindung löschen
 - `String[] list()`
Bindungen abfragen

- Woher bekommt man eigentlich die Referenz auf ein Registry-Objekt?
- Registry-Objekt der lokalen Maschine über statische Methoden der Klasse `java.rmi.Naming` zugänglich
 - `void java.rmi.Naming.bind(String name, Remote obj)`
 - `void java.rmi.Naming.rebind(String name, Remote obj)`
 - `Remote java.rmi.Naming.lookup(String name)`
 - `void java.rmi.Naming.unbind(String name)`
 - `String[] java.rmi.Naming.list()`

- Woher bekommt man eigentlich die Referenz auf ein Registry-Objekt?
- Registry-Objekte auf entfernten Maschinen über statische Methoden der Klasse `java.rmi.Naming` zugänglich
 - `Registry getRegistry()`
Lokale Registry auf Port
`java.rmi.registry.Registry.REGISTRY_PORT`
 - `Registry getRegistry(int port)`
Lokale Registry auf einem anderen Port
 - `Registry getRegistry(String host)`
Registry auf Rechner `host`
 - `Registry getRegistry(String host, int port)`
Registry auf Rechner `host`, Port `port`



Parametersemantik bei RMI

<i>Lokales Objektmodell</i>	<i>Verteiltes Objektmodell</i>
Aufruf an <i>Objekten</i>	Aufruf an <i>Interfaces</i>
Parameter und Ergebnisse als <i>Referenzen</i>	Parameter und Ergebnisse als <i>Kopien</i>
Alle Objekte fallen <i>zusammen</i> aus	<i>Einzelne</i> Objekte fallen aus
<i>Keine</i> Fehlersemantik	<i>Komplizierte</i> Fehlersemantik (Referenzintegrität, Netzfehler, Sicherheit etc.)
...	

Sortierobjekt

- Liste mit Standardmethoden sortieren:

```
import java.util.*;
public class ListSorter {
    public void sortList(List l) {
        Collections.sort(l);
    }
    public static void main(String[] argv) {
        ListSorter ls=new ListSorter();
        List list = new ArrayList();
        list.add("Tinky Winky");
        list.add("Dipsy");
        list.add("Laa-Laa");
        list.add("Po");
        ls.sortList(list);
        Iterator iter = list.iterator();
        while (iter.hasNext()) System.out.println(iter.next());
    }
}
```

Lokaler Aufruf

- Ausführung:
 - >java ListSorter
 - Dipsy
 - Laa-Laa
 - Po
 - Tinky Winky
- Alles wie erwartet
- Transformation in RMI-Objekt

Schnittstelle

```
import java.util.*;
public interface Sorter extends java.rmi.Remote {
    public void sortList(List l) throws java.rmi.RemoteException;
}
```

Server

```
import java.util.*;  
import java.net.*;  
import java.rmi.*;
```

```
public class SorterServer extends  
    java.rmi.server.UnicastRemoteObject  
    implements Sorter {
```

```
    ListSorter ls = new ListSorter();
```

```
    public SorterServer() throws java.rmi.RemoteException { }
```

```
    public void sortList(List l) throws java.rmi.RemoteException {  
        ls.sortList(l);  
    }
```

```
public static void main(String[] argv) {
    System.setSecurityManager(new RMI SecurityManager());
    try {
        SorterServer ss = new SorterServer();
        try {
            Naming.bind("rmi://" +
                (InetAddress.getLocalHost()).getHostName() + "/Sorter",ss);
        } catch (Exception e) {
            Naming.rebind("rmi://" +
                (InetAddress.getLocalHost()).getHostName() +
                "/Sorter",ss);
        }
    } catch (Exception e) {
        System.err.println(e.getMessage());
    }
}
```

```
import java.util.*; import java.rmi.*; import java.net.*;
public class SorterClient {
    public static void main(String[] argv) {
        try {
            Sorter ls= (Sorter)
                java.rmi.Naming.lookup("rmi://" +
                    (InetAddress.getLocalHost()).getHostName() + "/Sorter");
            List list = new ArrayList();
            list.add("Tinki Winki");      list.add("Dipsy");
            list.add("Lala");              list.add("Po");
            ls.sortList(list);
            Iterator iter = list.iterator();
            while (iter.hasNext())
                System.out.println(iter.next());
        } catch (Exception e) {System.err.println(e.getMessage());}
    }
}
```

Ausführung

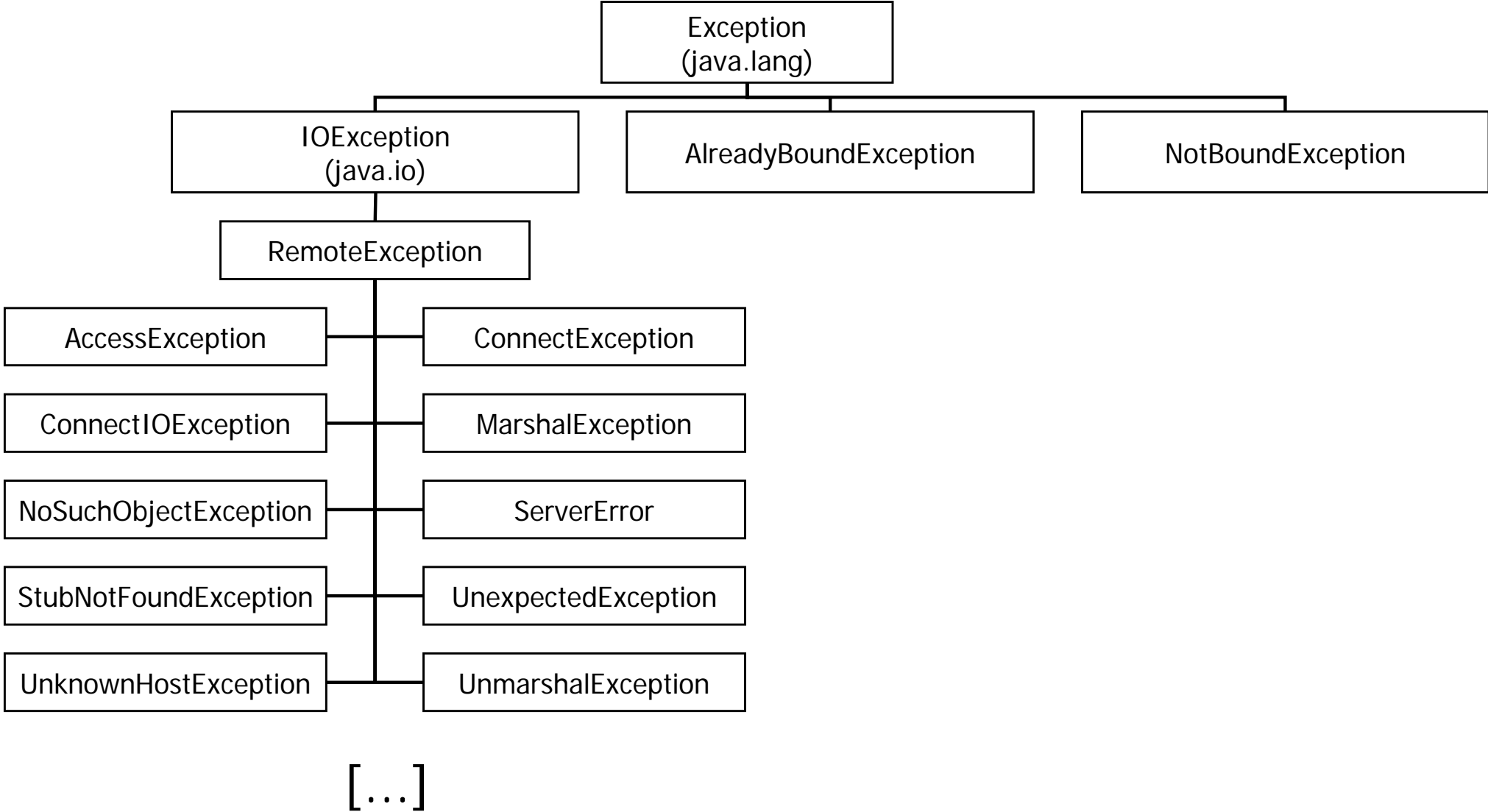
- Ausführung
 - > java SorterClient
 - Tinky Winky
 - Dipsy
 - Laa Laa
 - Po
- Feld des Klienten wird *nicht* sortiert!
- Es wird die *Kopie* dieses Feldes sortiert die beim Server ankommt
- Diese Kopie wird nach Sortieren irgendwann gelöscht
- Sie wird *nicht* zum Klienten übermittelt



RMI Fehler

<i>Lokales Objektmodell</i>	<i>Verteiltes Objektmodell</i>
Aufruf an <i>Objekten</i>	Aufruf an <i>Interfaces</i>
Parameter und Ergebnisse als <i>Referenzen</i>	Parameter und Ergebnisse als <i>Kopien</i>
<i>Alle Objekte fallen zusammen aus</i>	<i>Einzelne Objekte fallen aus</i>
<i>Keine Fehlersemantik</i>	<i>Komplizierte Fehlersemantik (Referenzintegrität, Netzfehler, Sicherheit etc.)</i>
...	

Ausnahmen aus java.rmi etc.



- `java.rmi.RemoteException`
 - Oberklasse für RMI Fehler
 - Enthält Feld in dem auf ursächliche Exception verwiesen wird
- `java.rmi.NotBoundException`
 - Unter angefragtem Name ist kein Objekt gebunden
- `java.rmi.UnknownHostException`
 - Angabe eines Rechnernamens kann nicht aufgelöst werden
- `java.rmi.AlreadyBoundException`
 - Ein Registry-Name ist schon an ein Objekt gebunden

- `java.rmi.ConnectException`
 - Entfernter Rechner nimmt keine Verbindungen an
- `java.rmi.ConnectIOException`
 - Beim Kontaktieren eines entfernten Rechners tritt ein Fehler auf
- `java.rmi.StubNotFoundException`
 - Bei Anmeldung eines Objekts in der Registry oder beim Aufruf kann der Stub nicht ermittelt werden (nicht im CLASSPATH, nicht richtig angemeldet, nicht einzurichten)
- `java.rmi.MarshalException`
 - Umwandeln von Aufruf in Netzrepräsentation scheitert
- `java.rmi.UnmarshalException`
 - Umwandeln von Aufruf aus Netzrepräsentation scheitert
- `java.rmi.NoSuchObjectException`
 - Objekt an dem eine Methode aufgerufen werden existiert nicht mehr

- `java.rmi.ServerError`
 - Fehler bei der entfernten Methodenausführung (ermittelbar im `ServerError` Objekt)
- `java.rmi.ServerException`
 - Ausnahme bei der entfernten Methodenausführung (ermittelbar im `ServerException` Objekt)
- `java.rmi.ServerRuntimeException`
 - Laufzeitausnahme bei der entfernten Methodenausführung (ermittelbar im `ServerException` Objekt)

- `java.rmi.UnexpectedException`
 - Während der Rückkehr (also nach Ausführung) einer Methode tritt beim Server oder beim Klienten eine Ausnahme auf (ermittelbar im `UnexpectedException` Objekt)
 - Unerwartet: Nicht in der Signatur der Methode deklariert
- `java.rmi.AccessException`
 - Nicht erlaubte Operation mit den Schnittstellen `Naming` oder `Registry`
 - Schreibende `Registry`-Zugriffe (`bind`, `rebind`, `unbind`) sind nur von der gleichen Maschine aus erlaubt
- `java.rmi.RMI SecurityException`
 - Sicherheitsregeln werden während Ausführung verletzt
 - Siehe später

Serverfehler I

- Das `java.rmi.server` Paket enthält die Grundfunktionalitäten für RMI Server-Objekte
 - Skeletons nutzen Klassen
 - `RemoteServer` ist Hauptklasse der RMI Server-Objekte
 - `UnicastRemoteObject` (bislang einzige) Unterklasse davon
- `java.rmi.server.ExportException`
 - Server-Objekt soll an einem schon benutzten Port exportiert werden
- `java.rmi.server.ServerCloneException`
 - Beim Duplizieren eines Server-Objekts tritt ein Fehler auf
- `java.rmi.server.ServerNotActiveException`
 - Server fragt nach dem Klienten außerhalb der Ausführung eines RMI-Aufrufs

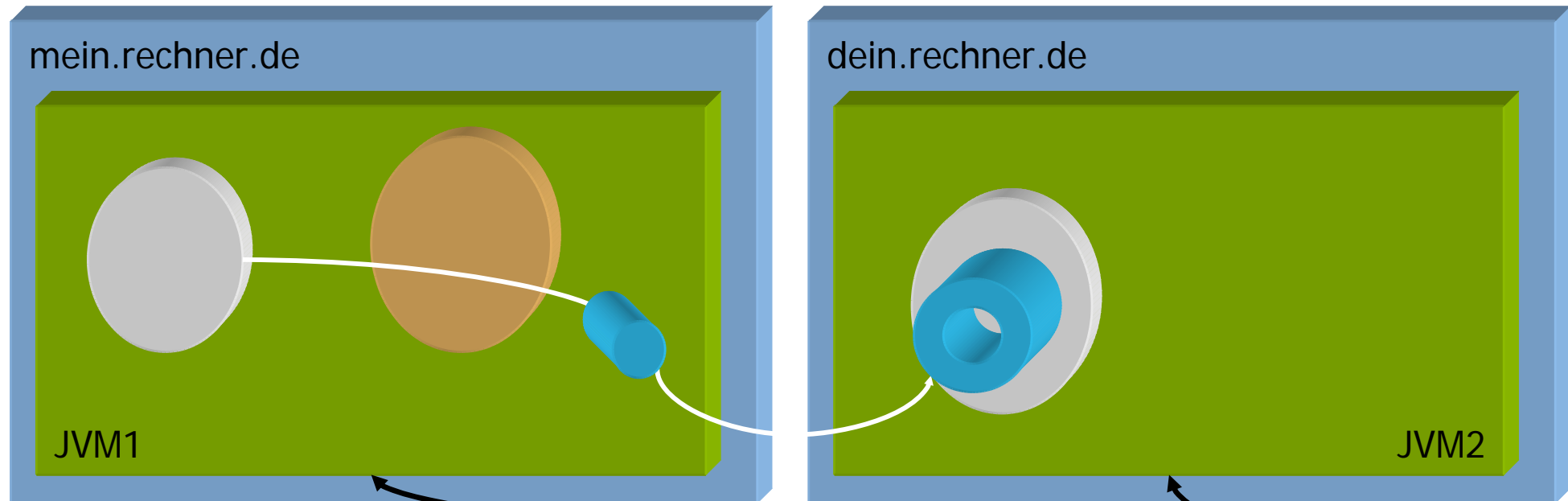
Serverfehler II

- `java.rmi.server.SkeletonNotFoundException`
 - Skeleton eines Objekts ist nicht auffindbar
- `java.rmi.server.SkeletonMismatchException`
 - Skeleton eines Objekts passt nicht (mehr) zum entfernten Stub bei einem Aufruf
 - Mittel: Hashcode über Stub- und Skeleton-Implementierungen
 - Wird an Clienten als `ServerException` weitergeleitet ist nicht auffindbar
- `java.rmi.server.SocketSecurityException`
 - Sicherheitsrichtlinien für Sockets werden beim Export eines Objekts verletzt

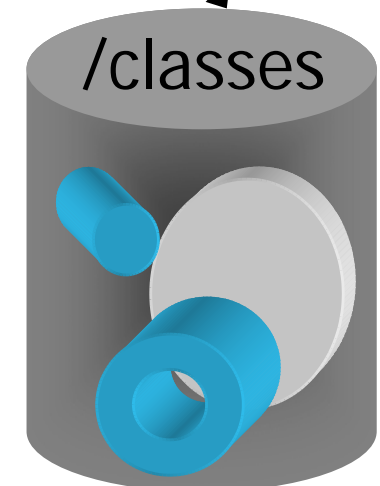


Code nachladen

Code nachladen

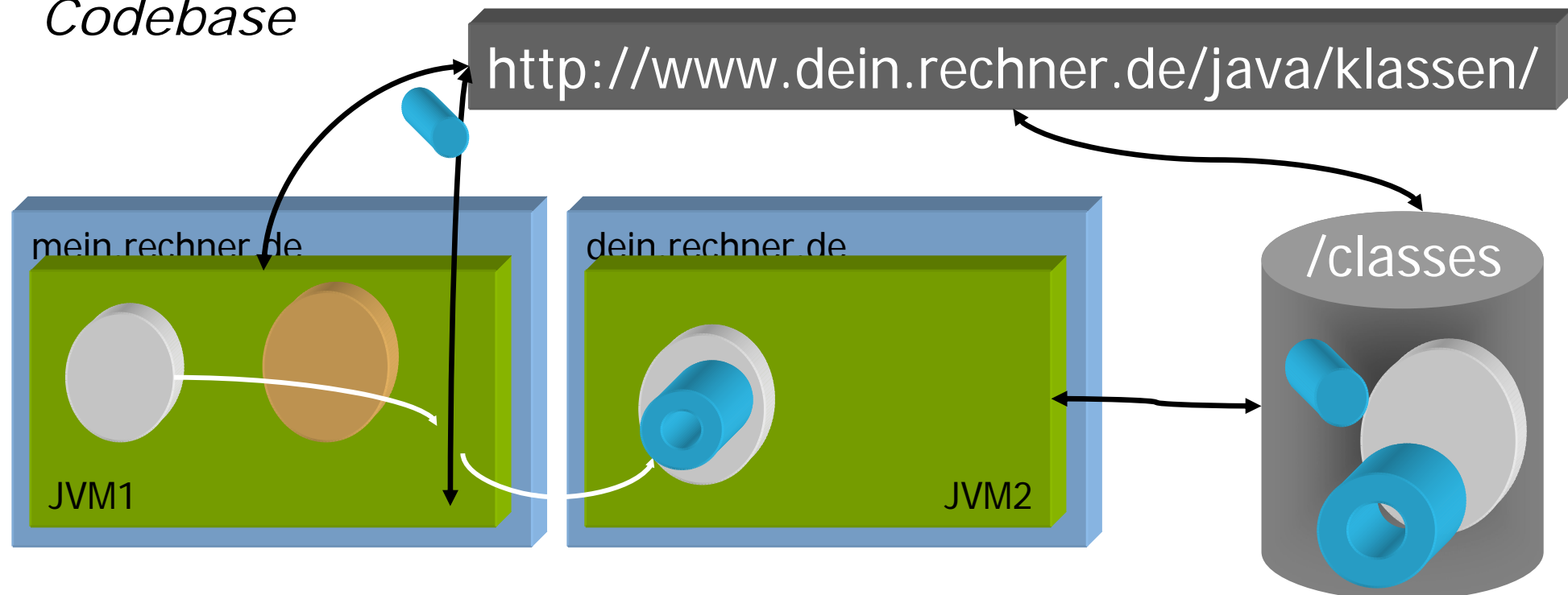


- Aufrufende JVM1 muss `ServerKlasse_stub.class` laden
- Wenn JVM1 und JVM2 im gleichen Dateisystem arbeiten, müssen sie entsprechend den `CLASSPATH` gesetzt haben



Nachladen über Web-Server

- Wenn JVM1 und JVM2 in getrennten Dateisystemen arbeiten nutzt CLASSPATH nichts
- ServerKlasse_stub.class muss über das Netz nachgeladen werden
- Dies geschieht durch Angabe einer Basis-URL, der *Codebase*



Die Codebase festlegen

- Eigenschaft `java.rmi.server.codebase` enthält URL der Codebase
- Beim Start des Serverobjekts festlegen:

```
>java -Djava.rmi.server.codebase=  
    http://www.dein.rechner.de/java/klassen/  
    CounterServer
```

- Im Programm festlegen:
`System.setProperty("java.rmi.server.codebase",codebase);`
- Codebase wird beim Registry-Eintrag vermerkt und beim lookup an Clienten übermittelt
- Codebase
 - CLASSPATH bezeichnet „lokale“ Codebase
 - `java.rmi.server.codebase` entfernte Codebase

Client auch mit Codebase

- Auch Client kann Codebase anbieten (müssen):
 - Server-Objekt bietet void $m(T_1 t_1)$ an
 - Client-Objekt ruft $m(t_2)$ auf mit T_2 als Unterklasse von T_1
 - T2.class muss geladen werden
 - Vergleichsweise mit Interfaces
- Also auch

```
>java -Djava.rmi.server.codebase=  
http://www.mein.rechner.de/classes/ AClient
```

- Codebase kann auch
 - jar-File sein
 - an mehreren Orten liegen
 - `java.rmi.server.codebase=`
"http://www.mein.rechner.de/classes/
http://www.dein.rechner.de/java/klassen/
http://www.unser.rechner/unsereklasse.jar"

Reale Problematik: Firewall

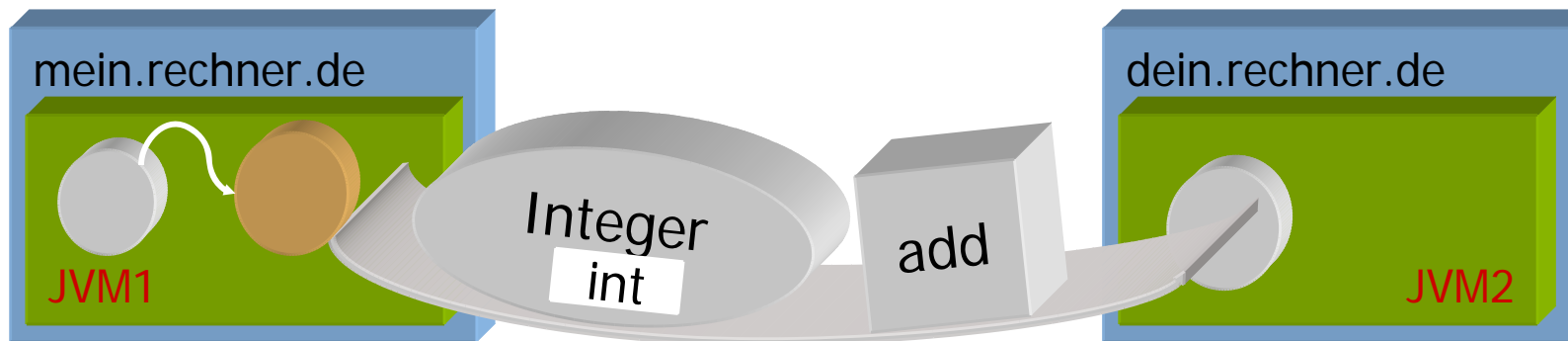
- Zum Nachladen von Code werden eigene Ports verwendet
- Diese können durch Firewalls blockiert werden
- Ports werden dynamisch festgelegt:
Firewall kann nicht konfiguriert werden
- Abhilfe:
 - Tunneling über HTTP
 - RMI-Transport wird über HTTP abgewickelt
 - java-rmi.cgi Skript muss auf Web-Server installiert sein
- Konfigurationsmöglichkeiten sprengen Umfang der Vorlesung
- <http://java.sun.com/developer/onlineTraining/rmi/RMI.html#RMISoftwareInstallation>



Serialisierung und Sicherheit

Objektserialisierung

- Parameter und Ergebnisse müssen übertragen werden:
(Konzeptionelle Darstellung, die Klassen werden ja nachgeladen und nicht mitgeschickt)






- Das Erstellen einer seriellen Repräsentation eines Objekts ist die Serialisierung (ggfs: Deserialisierung)
- Objektserialisierung notwendig zum Versenden oder Speichern von *Objekten*
- Instanzwerte + Klasse = Objektrepräsentation

Automatische Serialisierung

- Beispiel:

```
class Person {  
    String name;  
    Street adress;  
}
```

```
class Street {  
    String name;  
    int number;  
}
```

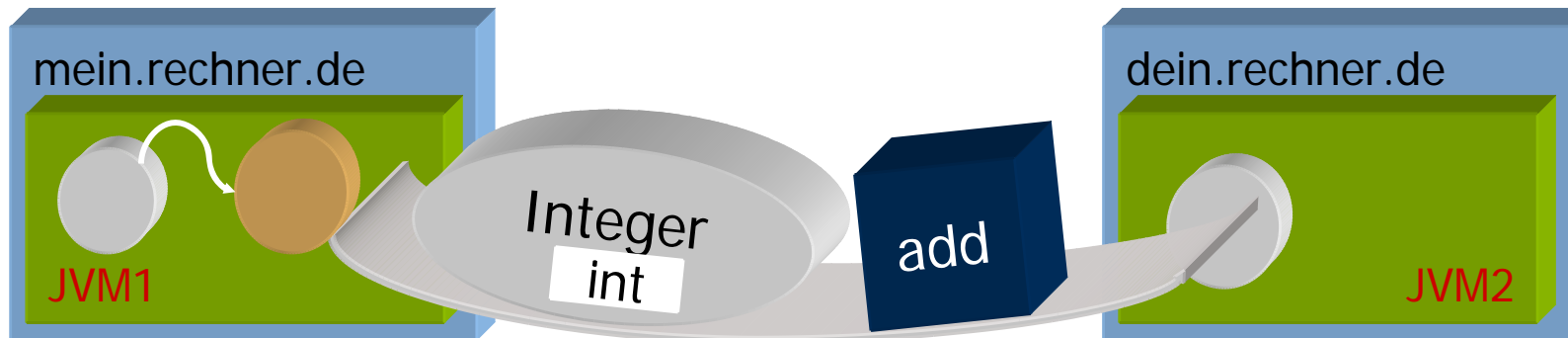
- Serialisiert:   
- Automatisch, wenn Klassen die Schnittstelle [java.io.Serializable](#) implementieren (lassen)
- `rmic` erzeugt Code für Serialisierung und Deserialisierung durch Analyse der Klassendefinitionen

Serialisierung

- **implements Serializable** bedeutet: Die automatische Serialisierung kann angewandt werden
- Wird von den meisten vordefinierten Klassen unterstützt, wenn möglich
- Gegenbeispiel: Ströme
- Schreibt man RMI Objekte und bewegt Objekte als Parameter oder Ergebnis, dann müssen diese Objekte als **Serializable** markiert sein
- Fehlerquelle: rmic hat Serialisierungscode erzeugt und man ändert danach das Objekt-Layout: Ergibt Marshalling Fehler zur Laufzeit
- Falls eigenes Marshalling notwendig, Definition von
 - `private void writeObject(java.io.ObjectOutputStream out) throws IOException`
 - `private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;`

Serialisierung: Klassen nachladen

- Objekte = Daten und Verhalten
- Serialisierte Java-Objekte: Datenstrom + Klasse (konzeptionell)



- Klassen zu übermittelten Objekten nachladen, falls nicht
 - vorher nachgeladen
 - vorher schon vorhanden (java.* Klassen)
- Bedrohung durch Angreifer:
 - Bildet Unterklasse von Street, ändert dabei toString()
 - Erzeugt Objekt davon
 - Übergibt Objekt als Argument beim Methodenaufruf
 - Beim Aufruf von toString() dort wird geänderter Code ausgeführt
 - Mit allen Rechten des RMI Objekts

Serialisierung: Klassen nachladen

- *SecurityManager* wird vor sicherheitsrelevanten Aktionen in verschiedenen Gruppen (Dateien, Sockets, GUI etc.) gefragt
- Ist Objekt mit Methoden wie:
 - void checkAccept(String host, int port)
 - void checkAccess(Thread t)
 - void checkAccess(ThreadGroup g)
 - void checkAwtEventQueueAccess()
 - void checkConnect(String host, int port)
 - void checkConnect(String host, int port, Object context)
 - void checkCreateClassLoader()
 - void checkDelete(String file)
 - ...
- Methoden werfen `java.lang.SecurityException` falls Ausführung nicht zugelassen

Security Manager

- Installation eines Security Managers mit der Methode `void System.setSecurityManager(SecurityManager s)`
- Fragt eventuell schon vorhandenen SecurityManager, ob das erlaubt ist
- RMI Classloader lädt Klassen von entfernten Rechnern nur dann nach, wenn SecurityManager installiert ist
- SecurityManager ist abstrakte Klasse
 - Im JDK mitgeliefert: RMI SecurityManager
`System.setSecurityManager(new RMI SecurityManager());`
 - In Browsern: Eigener SecurityManager

Policies

- Policies definieren im Detail, was erlaubt ist
- Securitymanager verwenden Policies um Rechte zu ermitteln
- Policies in einer Policy-Datei definiert
- Beispielausschnitt:

```
// Standard extensions get all permissions by default
grant codeBase "file:${java.home}/lib/ext/*" {
    permission java.security.AllPermission;
};
// default permissions granted to all domains
grant {
    // allows anyone to listen on un-privileged ports
    permission java.net.SocketPermission "localhost:1024-", "listen";
    // "standard" properties that can be read by anyone
    permission java.util.PropertyPermission "java.version", "read";
    ...
}
```

Erlaubnis für Socketnutzung

- Damit RMI arbeiten kann, muss es Socket-Verbindungen öffnen dürfen
- Das ist eventuell durch die lokal installierte Sicherheitspolicy verboten
 - Globale Policy-Definition bei *java.home/lib/security/java.policy*
 - Nutzerspezifische Policy-Definition bei *user.home/.java.policy*
 - `System.out.println(System.getProperty("user.home"));`
`System.out.println(System.getProperty("java.home"));`
 - C:\Documents and Settings\tolk
C:\Program Files\Java\j2re1.4.2_04

Erlaubnis für Socketnutzung

- Eintrag in .java.policy

```
grant { permission java.net.SocketPermission  
    "*", "accept,connect,listen,resolve";  
};
```

läßt beliebige Socketverbindungen zu

- Falls in anderer Datei:
`java -Djava.security.policy=/meindir/meinepolicy`
- Alternative: policytool mit GUI



Zusammenfassung

- Verteilte Objekte / RMI
 - Verteilte Objekte haben anderes Verhalten als lokale
 - Kommunikation über Schnittstellen/Proxy
- Objektreferenzen
 - rmiregistry als Objektverzeichnis
- Parametersemantik
 - Parameter als Kopie übergeben
- RMI Fehler
- Code nachladen
 - Klassencode von Webserver
 - Codebase Eigenschaft
- Serialisierung und Sicherheit
 - Serialisierbarkeit von Objekten
 - Policies

Literatur

- Sun. Java Remote Method Invocation Specification.
<http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html>
- Java Remote Method Invocation Homepage
<http://java.sun.com/products/jdk/rmi/>
- Dynamic code downloading using RMI
<http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html>
- Default Policy Implementation and Policy File Syntax
<http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html>