Freie Universität Berlin
Klaus-Peter Löhr

# 8 Java RMI

# 8.1 Fernaufrufbare Objekte

- Private und öffentliche Objekte können *fernaufrufbar* sein.
- Klasse und Vertreterklasse implementieren *gleiche Schnittstelle*.
- *Keine Fernerzeugung* von Objekten.
- *Mäßige* Verteilungsabstraktion

```
interface Counter {                    // lokal, ohne RMI
int inc(int i);
int value();
}

public class CounterImpl implements Counter {
        int c = 0;
public int inc(int i) { return c += i; }
public int value()    { return c; }
public int dec(int i) { return c -= i; }

}
```

```
import java.rmi.*;

interface Counter extends Remote {   // mit RMI
int inc(int i) throws RemoteException;
int value()    throws RemoteException;
}

public class CounterImpl implements Counter {
        int c = 0;
public int inc(int i) { return c += i; }
public int value()    { return c; }
public int dec(int i) { return c -= i; }

}
```

*Achtung - evtl. nichtsequentielle* Benutzung solcher Objekte
(hier der Einfachheit halber ignoriert)

- Ein mit `new` erzeugtes Objekt der Klasse `CounterImpl` ist lokal aufrufbar - *und kann zusätzlich fernaufrufbar gemacht werden („remote object").*

- Letzteres gelingt nur, wenn die Klasse ein `Remote` Interface implementiert, bei deren Methoden `RemoteException`s vereinbart sind (für verteilungsbedingte Fehler!).
  `Remote` ist ein leeres, sogenanntes *„marker interface"*.

- *Beachte:* Die Nutzen der Trennung von Schnittstelle und Implementierung wird bei Fernaufrufen besonders augenfällig. Ein Klient muss natürlich die Schnittstelle kennen. Die implementierende Klasse ist aber nicht nur unbekannt - *sie ist womöglich gar nicht lokal vorhanden.*

*Entfernte Benutzung* eines fernaufrufbaren Objekts:

```
public class Inc {
public static void main(String[] arg) {
        Counter x = .......    // Stub beschaffen !
        int i = x.inc(Integer.parseInt(arg[0]));
        System.out.println("counter is " + i);
        }
}
```

Beim Aufrufer ist die Klasse `CounterImpl` womöglich unbekannt!

Den Vertreter `x` erhält man durch einen Fernaufruf, der einen **Fernverweis** *(remote reference)* auf das Objekt liefert !

Objekt *fernaufrufbar* machen:

```
import java.rmi.server.UnicastRemoteObject;
```

Statische Methode `exportObject(x,port)`

- erzeugt für das `Remote` Objekt `x` Verwaltungsdaten,
- belegt dafür den Port `port` (irgendeinen, falls `0`).
- erzeugt für `x` einen *Vertreter* vom Typ `RemoteStub`,
- liefert einen Verweis auf diesen Vertreter.
- Für eine Sicht auf den Verweis gemäß Anwendungstyp ist eine explizite Typanpassung erforderlich, z.B.

```
(Counter)UnicastRemoteObject.exportObject(x,0)
```

Folgendes macht *alle Objekte* einer `Remote` Klasse fernaufrufbar:

```
class CounterImpl extends UnicastRemoteObject { ... }
```

# 8.2 Objektverzeichnis `rmiregistry`

- Start des Programms `rmiregistry` erzeugt einen Trägerprozess für ein **Register** öffentlicher Objekte:

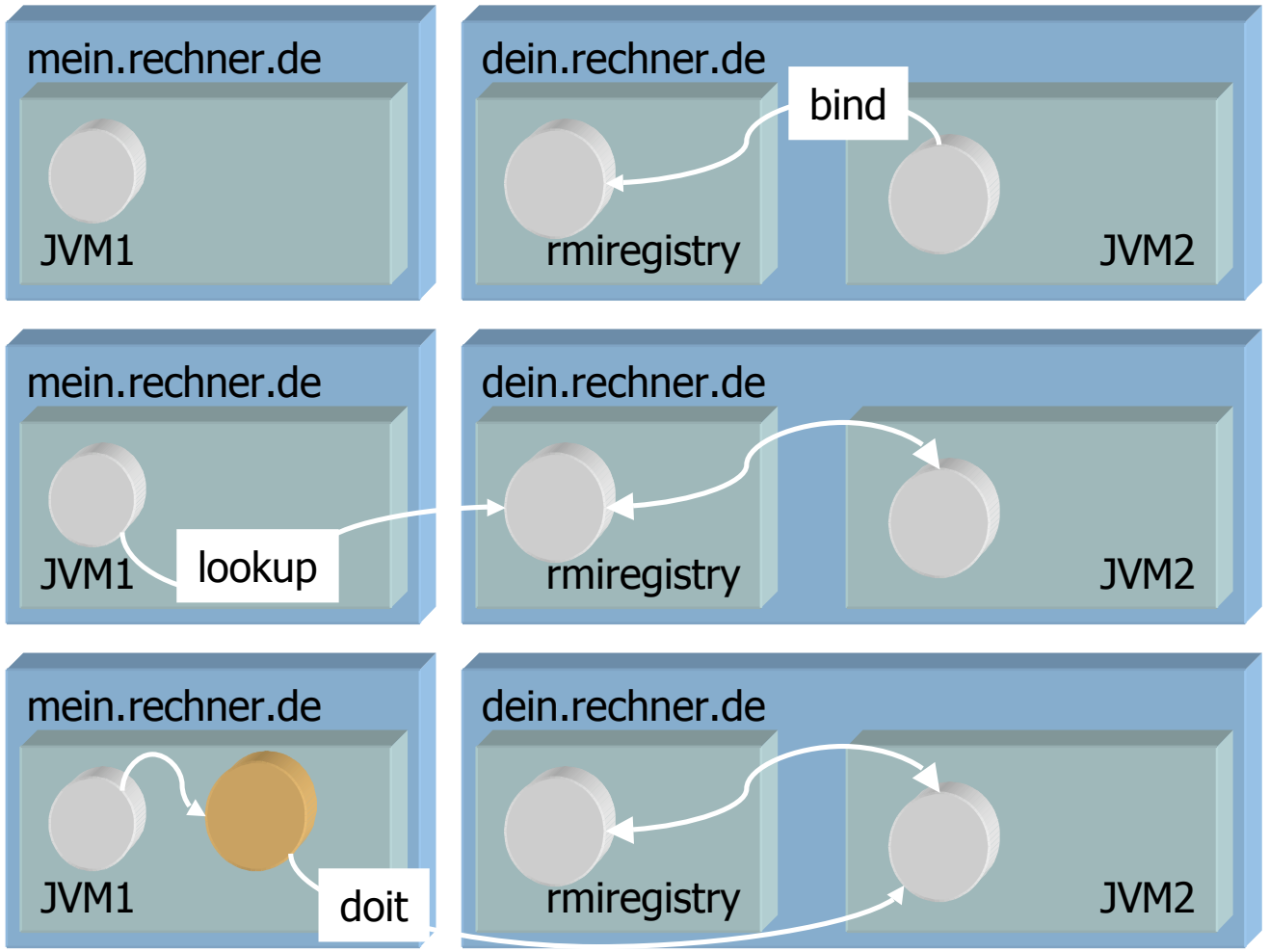
  `rmiregistry` <port> `&` (default port: 1099)

  *Nicht* 1099 verwenden! *Nicht* `kill` <procno> vergessen!

- Verwaltet werden ausschließlich *systemlokale* Objekte.

- Einträge haben die Form (Name, Fernverweis)

- Typische Verzeichnisoperationen `lookup, bind, rebind`


mein.rechner.de
JVM1
dein.rechner.de
bind
rmiregistry
JVM2
mein.rechner.de
JVM1
lookup
dein.rechner.de
rmiregistry
JVM2
mein.rechner.de
JVM1
doit
dein.rechner.de
rmiregistry
JVM2

```
import java.rmi.registry.Registry;
```

Eine `Remote` Schnittstelle von Objektverzeichnissen, mit Methoden `bind, rebind, lookup, .....`

```
import java.rmi.registry.LocateRegistry;
```

Statische Methode `getRegistry(host, port)` liefert einen *Client Stub* für den Fernaufruf eines (lokalen oder entfernten) `Registry`-Objekts in einem `rmiregistry`-Prozess.

Erzeugung und Bekanntmachung eines fernaufrufbaren Objekts:

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

public class CounterImpl implements Counter { int c = 0;
public int inc(int i) { return c += i; }
public int value()    { return c; }
public int dec(int i) { return c -= i; }
public static void main(String[] arg) throws Exception {
        Counter x = new CounterImpl();
        Counter stub = (Counter)UnicastRemoteObject.
                                exportObject(x,0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind("myCounter", stub);
        }
// main thread dies; hidden thread waits for invocation
}
```

Freie Universität Berlin

## Auffinden und Benutzen eines entfernten Objektes:

```
// usage:  java Inc host increment

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Inc {
public static void main(String[] arg) throws Exception {
        Registry registry = LocateRegistry.
                                getRegistry(arg[0]);
        Counter x = (Counter)registry.lookup("myCounter");
        int i = x.inc(Integer.parseInt(arg[1]));
        System.out.println("counter is " + i);
        }
}
```

Typanpassung - kann scheitern!

(Alternative: Klasse `java.rmi.Naming` verwenden)

Lokaler Test:

```
$ javac Counter.java
$ javac CounterImpl.java
$ javac Inc.java

$ java CounterImpl
Exception in thread "main" java.rmi.ConnectException:
Connection refused to host: 192.168.2.20 . . . . . .
```

[ *Register* läuft nicht! ]

```
$ rmiregistry &
[1] 8774
$
```

```
$ java Inc localhost 10
Exception ..... java.rmi.NotBoundException: myCounter
.............
```

[ CounterImpl läuft nicht ]

```
$ java CounterImpl &
[2] 8777

$ java Inc localhost 10
counter is 10
$ java Inc localhost 10
counter is 20
$ java Inc localhost -20
counter is 0

$ kill 8777
$ kill 8774
```

[ CounterImpl-Prozess löschen ]

[ rmiregistry-Prozess löschen ]

Test im Netz:

```
kpl@lounge: rmiregistry &
[1] 8774
kpl@lounge: java CounterImpl &
[2] 8777
kpl@lounge:
```

```
lohr@maclohr: java Inc lounge.mi.fu-berlin.de 10
counter is 10
lohr@maclohr:
```

```
fuzzy@xian: java Inc lounge 10
counter is 20
fuzzy@xian:
```

Was passiert hier?

```
Registry registry = LocateRegistry.getRegistry();
registry.rebind("myCounter", stub);
```
(S. 11)

Statische Methode `getRegistry(host, port)` liefert einen *Client Stub* für den Fernaufruf eines (lokalen oder entfernten) `Registry`-Objekts in einem `rmiregistry`-Prozess.

`rebind("myCounter", stub)` ruft das Register auf:
*Kopien* der Argumentobjekte werden übergeben (siehe →8.3),
und im Verzeichnis wird entsprechender Eintrag vorgenommen.

```
Counter x = (Counter)registry.lookup("myCounter");
```
(S. 12)

Dieser Fernaufruf liefert eine *Kopie* des im Register unter `"myCounter"` gespeicherten *Stubs*.

... und das Kleingedruckte:

- Die *Unmarshaling*-Routine im Register erwartet zum Herstellen des *Client Stub des Counter* aus der gepackten `rebind`-Nachricht die Schnittstelle `Counter.class` in seinem *Classpath*, d.h. `'.'` - wenn nicht anders festgelegt.
  (Sonst `ClassNotFoundException`)

- Später können Aufrufer und Aufgerufener durchaus mit nicht identischen Kopien der Schnittstelle `Counter.class` arbeiten. Mit kryptographischen Hash-Werten wird sichergestellt, dass die aufgerufene Methode vorhanden ist und die richtige Signatur hat.
  (Sonst `UnmarshalException: invalid method hash` )

```
ann@server:~ $ rmiregistry &
[1] 8774
ann@server:~ $
```

```
bob@server:~/classes $ ls
Counter.class
CounterImpl.class
bob@server:~/classes $ java CounterImpl &
[1] 8777
bob@server:~/classes $
Exception in thread "main" java.rmi.ServerException:
RemoteException occurred in server thread; nested exception is:
java.rmi.UnmarshalException: error unmarshalling arguments;
nested exception is: java.lang.ClassNotFoundException: Counter
.....
```

Dies wird wie folgt geheilt:

```
bob@server:~/classes $ ls
Counter.class
CounterImpl.class
bob@server:~/classes $ java \
>   -Djava.rmi.server.codebase=file:/Users/bob/classes/ \
>   CounterImpl &
[1] 8779
bob@server:~/classes $
```

Unverzichtbar!
Vgl. →8.4.

„Dem von `CounterImpl` versendeten Fernverweis wird die Codebasis der Schnittstelle beigegeben, damit der Empfänger die Schnittstelle `Counter.class` finden und damit einen Stub erzeugen kann."

*Dies ist gute RMI-Praxis,* wenn das Register von verschiedenen Personen für verschiedene Anwendungen eingesetzt werden soll.

# 8.3 Parameterübergabe

Nochmals eingeschärft:
Java kennt nur **einen** Parametermechanismus
Wertparameter *(call by value)*

Im Hinblick auf Fernaufrufe ist das
*einerseits gut*, weil es keine Variablenparameter gibt,
*andererseits schlecht*, weil es keine Ergebnisparameter gibt.

Außerdem ist die Allgegenwart von Verweisen ungünstig.

(Zur Erinnerung: 7.3.1)

„An argument to, or a return value from, a remote object can be any object that is *serializable*. This includes primitive types, remote objects, and non-remote objects that implement the `Serializable` interface." [RMI spec.]

Argumente/Ergebnisse von **Verweistypen**:

- Wenn das Argument/Ergebnis *fernaufrufbar* ist (S. 7!), wird ein Fernverweis übergeben.
- Wenn das Argument/Ergebnis *nicht fernaufrufbar* ist, aber `Serializable`, wird eine Kopie übergeben.
- Sonst wird der Fehler `MarshalException` gemeldet (bedingt durch `NotSerializableException`).

Entsprechend für **Komponenten** von Argumenten/Ergebnissen !

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

public class Par {
public static void main(String[] arg) throws Exception {
        Counter count = new CounterImpl();
        Counter stub  = (Counter)UnicastRemoteObject.
                                    exportObject(count,0);
        Registry registry = LocateRegistry.getRegistry(arg[0]);
        Param x = (Param)registry.lookup("param");
        int i = x.call(count);     // executes 'return c.inc(7);'
        int k = x.call(stub);      // executes 'return c.inc(7);'
        System.out.println("counter is " + i + " " + k + " " +
                           count.value() + " " + stub.value());
       System.exit(0);             // enforces program shutdown
   }
}
```

```
$ java Par localhost
counter is 7 14 14 14
$
```

*Nichtlokal* - evtl. Problem bei IP-Adressen-Umsetzung:

```
→ xian:  java ParamImpl &

→ home:  java Par xian.imp.fu-berlin.de
  Exception in thread "main" java.rmi.ServerException:
  RemoteException occurred in server thread;
  nested exception is: java.rmi.ConnectIOException:
        Exception creating connection to: 192.168.2.23;
  nested exception is: java.net.NoRouteToHostException:
        No route to host
```

Dies wird wie folgt geheilt:

meine IP-Adresse im VPN

```
home:  java -Djava.rmi.server.hostname=130.133.48.229 \
>           Par xian.imp.fu-berlin.de
counter is 7 14 14 14
home:
```

3-mal Kleingedrucktes zur Verteilungsabstraktion:

❶
```
Counter x1 = (Counter)registry.lookup("myCounter");
Counter x2 = (Counter)registry.lookup("myCounter");
```

Hier gilt *leider nicht* `x1 == x2` (da der Fernaufruf `lookup` ein neues Stub-Objekt liefert), wohl aber `x1.equals(x2)` - dank einer Redefinition von `equals` für Stub-Objekte.

❷ `object.method(node1, node2);` (7.3.1, S. 24)

RMI betrachtet `(node1, node2, ...)` als *ein* Geflecht !

„If two references to an object are passed from one JVM to another JVM in parameters (or in the return value) in a single remote method call and those references refer to the same object in the sending JVM, those references will refer to a single copy of the object in the receiving JVM."

❸ `synchronized(remobj) {...}`

Vorsicht! Hier wird der Vertreter gesperrt, nicht das Objekt!

... und die Fehlersemantik:

„If a `MarshalException` occurs during a remote method call, the call *may or may not* have reached the server. If the call did reach the server, parameters *may* have been deserialized. *A call may not be retransmitted after a* `MarshalException` *and reliably preserve ‚at most once' call semantics.*" (`java.rmi.MarshalException` API)

M.a.W.: Nicht nur gibt RMI *keine Garantien* bezüglich einer präzisen Fehlersemantik, der Benutzer hat auch bei der Ausnahmebehandlung keinerlei Möglichkeit, etwas Besseres zu realisieren.

# Zusammenfassung

- Die *Verteilungsabstraktion* von RMI ist nur mäßig gut ausgeprägt: für die Realisierung von Fernaufrufen werden objektorientierte Techniken *explizit* eingesetzt (`Remote`, `RemoteException`, ...). Klassenmethoden sind grundsätzlich *nicht* fernaufrufbar. Es gibt *keine* Fernerzeugung privater Objekte.

- Keine Fernaufrufe ohne ein **Register**, das als initiales fernaufrufbares Objekt ansprechbar ist!

- Die im Register verzeichneten Objekte sind **öffentlich** - jeder kann sie über das Register erreichen.

- Die Semantik der Parameterübergabe ist *nicht unabhängig* davon, ob ein Aufruf „normal" oder über als Fernaufruf erfolgt.

- Bei der Behandlung eigener Anwendungsklassen, deren Objekte per Kopie übergeben werden sollen, muss man die Codeverwaltung und Sicherheitsfragen (→ 8.4, 8.5) im Blick behalten.

- Fernaufrufe über die Firewalls von Intranetzen hinweg werden in der Regel nicht zugelassen und erfordern besondere Maßnahmen.

- Und nie vergessen: öffentliche Objekte befinden sich in einer potentiell *nichtsequentiellen* Umgebung! (ALP 4 ist hier nicht nur nicht irrelevant, sondern wird sogar besonders relevant.)

# Quellen

Oracle Corp.: *RMI Specification.*
docs.oracle.com/javase/6/docs/platform/rmi/spec/rmiTOC.html

Oracle Corp.: *RMI Tutorial.*
docs.oracle.com/javase/tutorial/rmi/

W. Grosso: *Java RMI.* O'Reilly 2001.

und die Literatur auf www.inf.fu-berlin.de/lehre/WS12/alp5/literatur.html