

# Remote-Objekte



## Remote-Objekte

### Motivation

Grundarchitektur

Implementierung von Remote-Objekten

Parameterübergabe

Callbacks

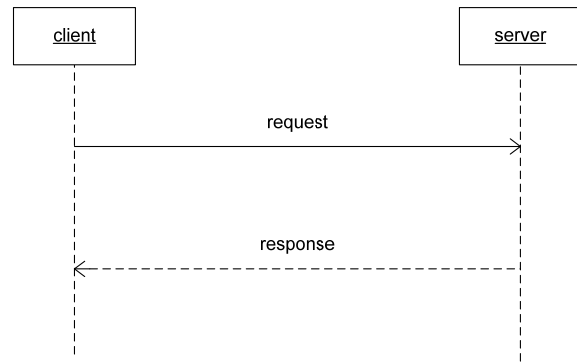
Distributed Garbage Collection

Verteilung und Nachladen von Code

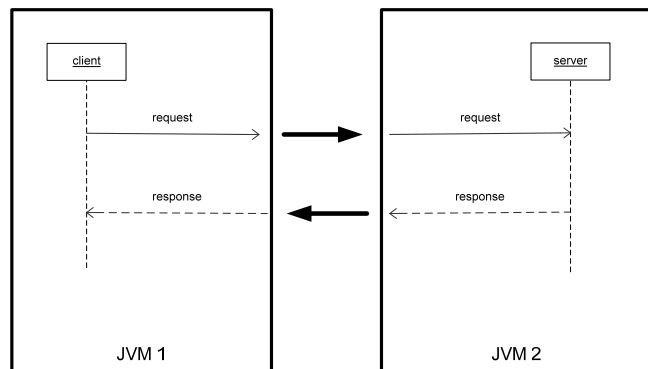


# Motivation

- Methodenaufwurf bei einer VM
  - Objekte innerhalb einer JVM



- Methodenaufwurf über VM Grenzen hinweg
  - Objekte sind auf mehreren JVMs (Rechnern) verteilt



# Probleme bei Remote-Objekten

- Objektreferenzen:
  - Was ist eine Remote-Referenz?
  - Wie finde ich ein Objekt?
  - Wie spreche ich Objekte an?
- Methodenaufwurf:
  - Wie wird die Sprungadresse für die Methode ermittelt?
- Parameterübergabe und Rückgabewerte:
  - Wie werden Parameterwerte und Rückgabewerte verschickt?
- Objekterzeugung:
  - Wie kann der Client ein Remote-Objekt erzeugen?
- Garbage Collection:
  - Wann kann ein Objekt am Server frei gegeben werden?
- Laden der Klassen:
  - Von wo wird die Klasse für ein vom Client benötigtes Objekt geladen?



# Remote-Objekte

Motivation

Grundarchitektur

Implementierung von Remote-Objekten

Parameterübergabe

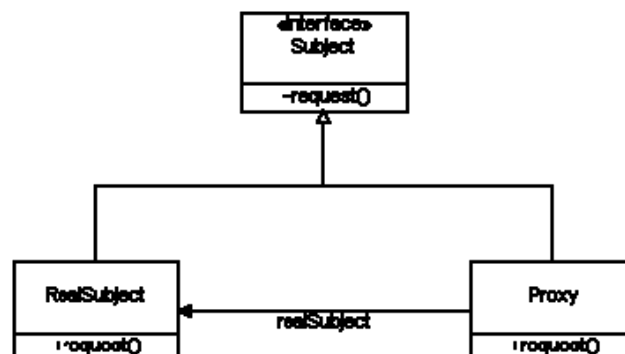
Callbacks

Distributed Garbage Collection

Verteilung und Nachladen von Code



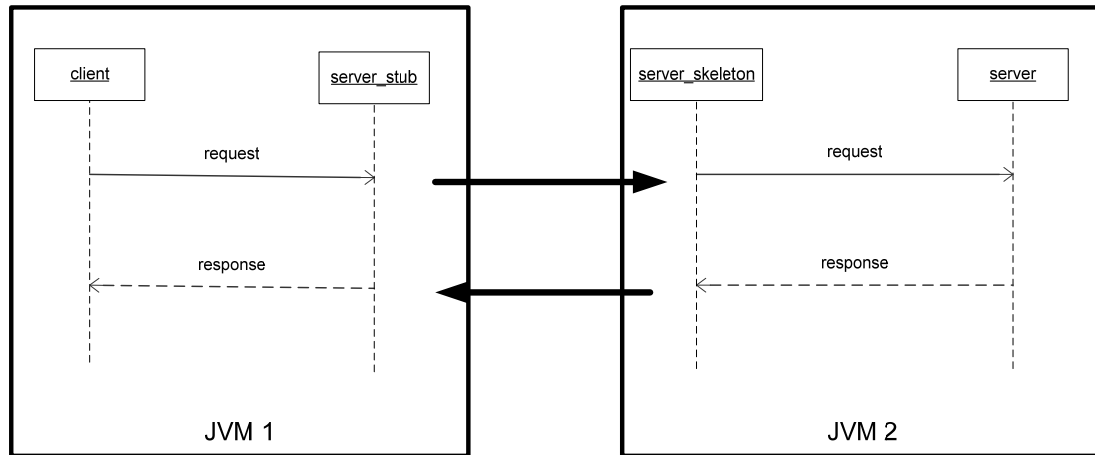
# Proxy-Pattern



- Interface:
  - definiert eine allgemeine Schnittstelle, die Client sieht und Server implementieren muss
- Implementierung für den Server
  - hat Interface entsprechend zu implementieren
- Proxy für den Client
  - wird vom Client angesprochen
  - implementiert die Schnittstelle, d.h. stellt sich dar wie ein wirkliches Objekt
  - delegiert alle Requests an die wirkliche Objektimplementierung



# Proxy-Pattern bei Remote-Objekten

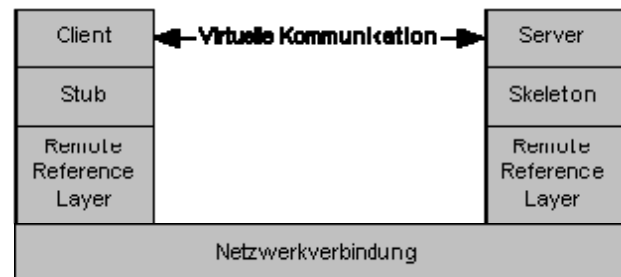


- Stub stellt das Proxy-Objekt auf der Seite des Client dar
- Proxy leitet die Anforderungen über die VM-Grenzen an Server weiter
- Skeleton empfängt die Requests und baut eigentlichen Request an den Server auf
- Server bedient den Request
- Ergebnis wird an Skeleton zurückgegeben
- Ergebnis wird über Netzwerk an Stub weitergeleitet
- Stub gibt das Ergebnis an Client zurück



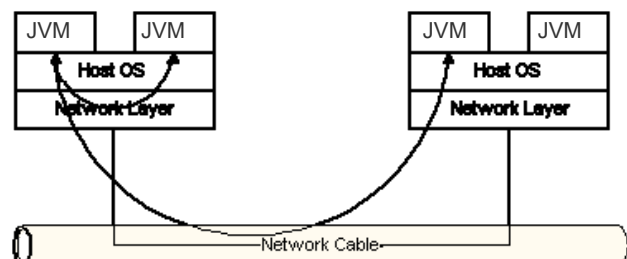
# Kommunikationsarchitektur

Client und Server  
Stubs und Skeletons  
Remote Reference Layer  
Netzwerkverbindung



## Netzwerkverbindung

- zwischen JVMs immer über Network Layer des Host OS

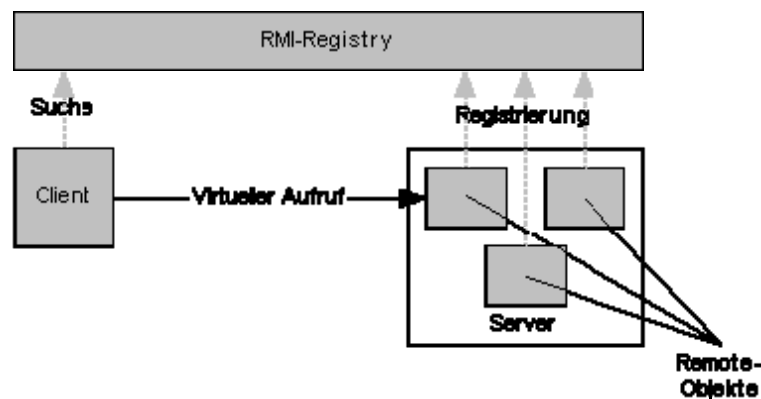


# Kommunikationsarchitektur: Aufgaben der Schichten

- Client und Server
  - sind die eigentlichen Anwendungsobjekte
  - kommunizieren über virtuelle Kommunikation
- Stubs und Skeletons
  - Stub (**Achtung: ab Java 1.5 nicht mehr nötig; mittels Dynamic-Proxies gelöst**)
    - ist das Proxy-Objekt auf der Client-Seite und stellt sich wie Server-Objekt dar
    - leitet Requests des Client an die unteren RMI-Services weiter
  - Skeleton (**Achtung: ab Java 1.2 nicht mehr nötig!!!**)
    - empfängt die Requests von unteren RMI-Services und
    - setzt diese für Requests an Server-Objekt um
- Remote Reference Layer
  - verwaltet die Remote-Referenzen (Klasse RemoteRef) der Server-Objekte
  - RemoteRef bietet Methode invoke für Methodenaufrufe abzusetzen (wird von Stubs verwendet)
  - dient zur Registrierung von Remote-Objekts
- Netzwerkverbindung
  - eigentliche Netzwerkverbindung über TCP/IP
  - stream-basierte Verbindung
  - standardmäßig auf Port 1099
  - Kommunikationsprotokoll oberhalb TCP/IP (nicht standardisiert)



# Objektregistrierung und Objektsuche



- RMI ermöglicht
  - Registrierung von Remote-Objekten durch Server mit einer RMI-URL
  - Aufsuchen von Remote-Objekten durch Client
- Dazu dienen
  - RMI-Registrierungsprogramm **rmiregistry**
  - RMI-Service Klasse **Naming**



## Objektregistrierung und Objektsuche: Vorgehen

- Über Service-Klasse `Naming` wird mit `bind` oder `rebind` ein Server-Objekt registriert. Es muss dabei ein eindeutiger Name für das Server-Objekt vergeben werden.

```
try {
    BankManager bm = new BankManagerImpl ();
    Naming.rebind("BankService", bm);
}
...
```

Das Server-Objekt wird dabei unter der RMI-URL mit `rmi://<HostComputer>:1099/BankService` eingetragen.

- Damit hat ein Client eine eindeutige URL, um auf ein Server-Objekt zuzugreifen; dazu dient abermals die Service-Klasse `Naming`.

```
// Client
try {
    BankManager bm = Naming.lookup("rmi://bankserver.com/BankService");
    ...
}
...
```

default port 1099



## Remote-Objekte

Motivation

Grundarchitektur

Implementierung von Remote-Objekten

Parameterübergabe

Callbacks

Distributed Garbage Collection

Verteilung und Nachladen von Code



# Realisierung eines Remote-Objekts

## Interfaces und Klassen

### Interface Spezifikation für Remote-Objekt

- muss `java.rmi.Remote` erweitern
- definiert die sichtbaren Methoden des Remote-Objekts
- jede Methode muss eine `RemoteException` werfen

### Server-Objekt

- muss Interface implementieren
- muss an das RMI System exportiert werden

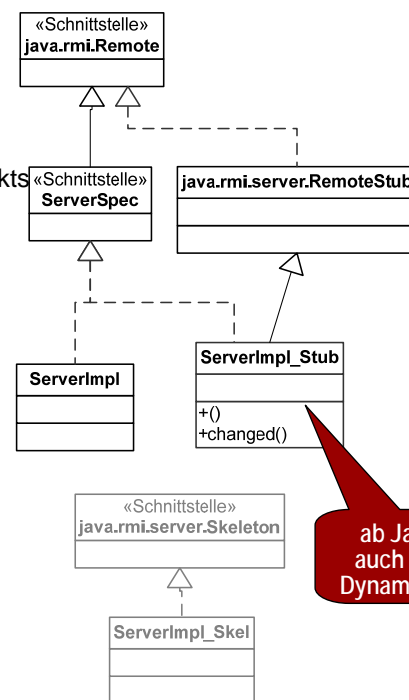
### Stub-Objekt

(kann ab Java 1.5 durch Dynamic-Proxy ersetzt werden)

- wird mit `rmi c` erzeugt
- implementiert Interface

### Skeleton-Objekt (nur bei Java 1.1 verwendet)

- wird mit `rmi c` erzeugt
- erweitert Basisklasse `java.rmi.server.Skeleton`



# Beispiel Calculator

Im folgenden Beispiel wird anhand eines einfachen Calculator-Servers die Realisierung eines Client-Server-Programms veranschaulicht

Folgende Schritte sind dabei durchzuführen:

- Erstellen
  - Definition eines Interfaces für das Remote-Objekt
  - Implementierung des Interfaces für den Server
  - Generierung von Stub- und Skeleton-Klassen mit `rmi c`
  - Implementierung des Server-Programms, welches das Server-Objekt anlegt und bei der RMIRegistry registriert
  - Realisierung eines Client-Programms, welches auf das Server-Objekt zugreift und dieses verwendet
- Starten
  - Starten des `rmi registry`
  - Starten des Server-Programms
  - Starten des Client-Programms



## Beispiel Calculator: Interface *Calculator*

- Interface `Calculator` definiert eine Reihe von Methoden für die Grundrechnungsarten

```
public interface Calculator extends java.rmi.Remote {  
  
    public long add(long a, long b)  
        throws java.rmi.RemoteException;  
  
    public long sub(long a, long b)  
        throws java.rmi.RemoteException;  
  
    public long mul(long a, long b)  
        throws java.rmi.RemoteException;  
  
    public long div(long a, long b)  
        throws java.rmi.RemoteException;  
  
}
```



## Beispiel Calculator: Implementierung `CalculatorImpl`

- Klasse `CalculatorImpl` stellt eine entsprechende Implementierung von `Calculator` bereit

```
public class CalculatorImpl implements Calculator {  
  
    public long add(long a, long b) throws java.rmi.RemoteException {  
        return a + b;  
    }  
  
    public long sub(long a, long b) throws java.rmi.RemoteException {  
        return a - b;  
    }  
  
    public long mul(long a, long b) throws java.rmi.RemoteException {  
        return a * b;  
    }  
  
    public long div(long a, long b) throws java.rmi.RemoteException {  
        return a / b;  
    }  
  
}
```



# Beispiel Calculator: Generierung von Stub und Skeleton

- Stub- und Skeleton-Klassen müssen nicht geschrieben werden
- Sie können durch das Programm

**rmi c**

automatisch generiert werden

- **rmi c** wird mit Server-Implementierungsklasse aufgerufen

Microsoft Windows XP

```
> rmi c -keep CalculatorImpl
```

**rmi c <options> <class names>**

where <options> includes:

```
-keep Do not delete intermediate generated source files
-g Generate debugging info
-depend Recompile out-of-date files recursively
-nowarn Generate no warnings
-verbose Output messages about what the compiler is doing
-classpath <path> Specify where to find input source and class files
-d <directory> Specify where to place generated class files
-J<runtime flag> Pass argument to the java interpreter
-v1.1 Create stubs/skeletons for JDK 1.1 stub protocol version
-vcompat (default) Create stubs/skeletons compatible with both JDK 1.1 and Java 2 stub protocol versions
-v1.2 Create stubs for Java 2 stub protocol version only
```

```
public final class CalculatorImpl_Stub
    extends java.rmi.server.RemoteStub
    implements Calculator, java.rmi.Remote
{
    ...
}
```

```
public final class CalculatorImpl_Skel
    implements java.rmi.server.Skeleton
{
    ...
}
```

Achtung: kann ab Java 1.5 gänzlich entfallen!



# Beispiel Calculator: Server-Programm

- Server-Programm hat folgende Aufgaben:
  - erzeugen des Server-Objekts
  - exportieren des Server-Objekts an das RMI
  - Registrieren des Server-Objekts unter einem RMI-URL mittels Klasse Naming

```
import java.rmi.Naming;

public class CalculatorServer {
    public CalculatorServer() {
        try {
            Calculator calc = new CalculatorImpl();
            RemoteStub calcStub = UniCastRemoteObject.exportObject(calc);
            Naming.rebind("rmi://localhost:1099/CalculatorService", calcStub);
        } catch (Exception e) {
            System.out.println("Trouble: " + e);
        }
    }

    public static void main(String args[]) {
        new CalculatorServer();
    }
}
```



## Exportieren von Server-Objekten

- Server-Objekte müssen bei ihrer Erzeugung an das RMI exportiert (bekannt gemacht) werden
- Dies kann auf zwei Arten passieren
  1. mit static-Methode `exportObject` von Klasse `UnicastRemoteObject` (wie im letzten Beispiel)

```
static RemoteStub exportObject(Remote obj) throws RemoteException  
static Remote exportObject(Remote obj, int port) throws RemoteException
```

Port 0 bedeutet, dass Port vom System gewählt wird.

2. durch Ableiten von Klasse `UnicastRemoteObject`, Implementierung eines Konstruktors der `RemoteException` und Aufruf des Konstruktors der Basisklasse `UnicastRemoteObject`

```
public class CalculatorImpl  
    extends java.rmi.server.UnicastRemoteObject implements Calculator {  
  
    // Implementations must have constructor with RemoteException  
    public CalculatorImpl() throws java.rmi.RemoteException {  
        super();  
        ...  
    }  
    ...  
}
```



## Beispiel Calculator: Client-Programm

- Client-Programm wird:
  - sich über den Naming-Service mit dem URL das Remote-Objekt holen
  - das Remote-Objekt verwenden (dabei direkt den Stub aufrufen)
  - eine Reihe von Exceptions abfangen

```
public class CalculatorClient {  
  
    public static void main(String[] args) {  
        try {  
            Calculator calc = (Calculator) Naming.lookup("rmi://localhost/CalculatorService");  
  
            System.out.println( calc.sub(4, 3) );  
            System.out.println( calc.add(4, 5) );  
            System.out.println( calc.mul(3, 6) );  
            System.out.println( calc.div(9, 3) );  
        } catch (MalformedURLException murl) {  
            ...  
        } catch (RemoteException re) {  
            ...  
        } catch (NotBoundException nbe) {  
            ...  
        }  
        catch (  
            ...  
        }  
    }  
}
```



## Beispiel Calculator: Starten der Programme

- Starten des RMIRegistry-Programms

```
Microsoft Windows XP  
> rmi registry
```

**Achtung:** Starten aus dem Verzeichnis des Java-Programms (andere Codebase siehe später)

- Starten des Server-Programms

```
> java CalculatorServer
```

- Starten des Client

```
> java CalculatorClient  
1  
9  
18  
3
```



## Alternative zu rmiregistry: java.rmi.registry.Registry

- Als einfache Alternative zum Start von `rmi registry` kann man mit `java.rmi.registry.Registry` arbeiten

### Server:

```
import java.rmi.registry.LocateRegistry;  
import java.rmi.registry.Registry;  
  
public class CalculatorServer {  
    public CalculatorServer() {  
        try {  
            Calculator c = new CalculatorImpl();  
            Registry reg = LocateRegistry.createRegistry(1099);  
            reg.bind("CalculatorService", c);  
            ...  
        }  
    }  
}
```

### Client:

```
public class CalculatorClient {  
  
    public static void main(String[] args) {  
        try {  
            Calculator c = (Calculator)  
                Naming.lookup("rmi://localhost:1099/CalculatorService");  
            ...  
        }  
    }  
}
```



## Alternative zu Naming: javax.naming.Context

- javax.naming enthält Naming-Services
  - javax.naming.Context - Interface für einen Naming-Context
  - javax.naming.InitialContext - Klasse für einen initialen Context

### Server:

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

public class CalculatorServer {
    public CalculatorServer() {
        try {
            Calculator c = new CalculatorImpl();
            Context context = new InitialContext();
            context.bind("CalculatorService", c);
            ...
        }
    }
}
```

### Client:

```
public class CalculatorClient {

    public static void main(String[] args) {
        try {
            Context context = new InitialContext();
            Calculator c = (Calculator)
                context.lookup("rmi://localhost:1099/CalculatorService");
            ...
        }
    }
}
```



## Remote-Objekte

### Exkurs: Dynamic Proxy



## Exkurs: Dynamic Proxy

- package java.lang.reflect
- dynamisch erzeugter Proxy implementiert Liste von Interfaces
  - mit statischer Methode `Proxy.newProxyInstance` erzeugt
- Proxy-Objekt hat gleiches Interface wie ursprüngliches Objekt, ruft aber InvocationHandler auf

```
public interface InvocationHandler {
    Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable
}

public class Proxy {
    public static Object newProxyInstance(
        ClassLoader loader,
        Class<?>[] interfaces,
        InvocationHandler h )
        throws IllegalArgumentException;

    public static Class<?> getProxyClass(ClassLoader loader,
        Class<?>... interfaces)
        throws IllegalArgumentException
    ...
}
```

### Anwendung:

```
Foo f= (Foo)Proxy.newProxyInstance(null, new Class[] {Foo.class}, handler);
```



## Beispiel Dynamic Proxy: TraceHandler (1)

- Beispiel: TraceHandler
  - TraceHandler realisiert InvocationHandler
  - zuerst Trace-Ausgabe
  - dann Aufruf der eigentlichen Methode mit den Argumenten

```
class TraceHandler implements InvocationHandler{
    public TraceHandler(Object t) {
        target = t;
    }

    public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable {
        System.out.print(target + "." + m.getName() + "(");
        if (args != null) {
            for (int i = 0; i < args.length; i++) {
                System.out.print(args[i]);
                if (i < args.length - 1)
                    System.out.print(", ");
            }
        }
        System.out.println(")");
        return m.invoke(target, args);
    }
    private Object target;
}
```

Ausgabe der  
Traceinfo

Aufruf der eigentlichen  
Methode



## Beispiel Dynamic Proxy: TraceHandler (2)

- Test mit Proxy für Integer-Werte
- mit Trace der compareTo-Methode des Comparable-Interfaces

```
public class ProxyTest {
    public static void main(String[] args) {
        Object[] elements = new Object[1000];

        // fill elements with proxies for the integers 1 ... 1000
        for (int i = 0; i < elements.length; i++) {
            Integer value = i + 1;
            Class[] interfaces = value.getClass().getInterfaces();
            InvocationHandler handler = new TraceHandler(value);
            Object proxy = Proxy.newProxyInstance(null, interfaces, handler);
            elements[i] = proxy;
        }

        Integer key = ...;
        // search for the key
        int result = Arrays.binarySearch(elements, key);

        // print match if found
        if (result >= 0)
            System.out.println(elements[result]);
    }
}
```

{ Comparable.class }

```
500.compareTo(547)
750.compareTo(547)
625.compareTo(547)
562.compareTo(547)
531.compareTo(547)
546.compareTo(547)
554.compareTo(547)
550.compareTo(547)
548.compareTo(547)
547.compareTo(547)
547.toString()
547
```



## Remote-Objekte

Motivation

Grundarchitektur

Implementierung von Remote-Objekten

Parameterübergabe

Callbacks

Distributed Garbage Collection

Verteilung und Nachladen von Code



# Parameterübergabe und Rückgabewerte

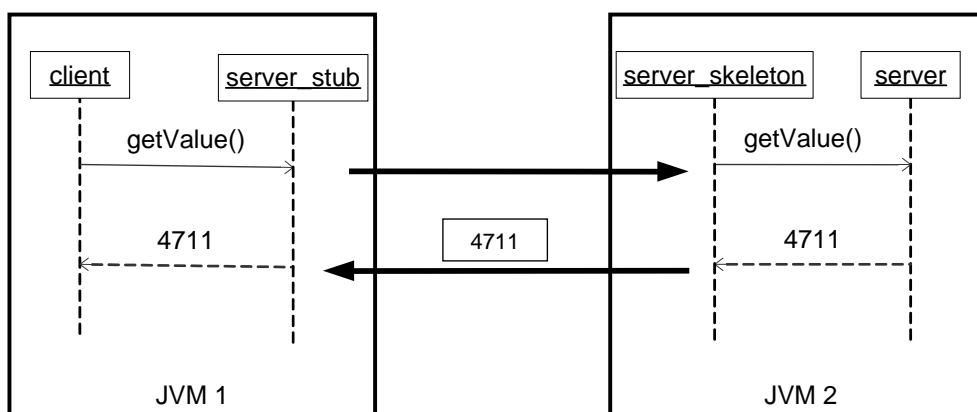
Bei RMI sind folgende Arten der Parameterübergabe und Rückgabewerte zu unterscheiden:

- **Basisdatentypen** (int, double, boolean, etc.)
  - **Werte** werden über das Netzwerk übertragen
- **Serialisierbare Objekte** (implementieren Interface `Serializable`)
  - Objekte werden serialisiert über das Netzwerk übertragen und auf der anderen Seite eine **Kopie** aufgebaut
- **Remote-Objekte** (implementieren Interface `Remote`)
  - Es wird auf der empfangenden Seite ein **Stub** für das Objekt aufgebaut
- **alle anderen** (nicht serialisierbar, nicht Remote)
  - diese **können nicht als Parameter** übergeben werden



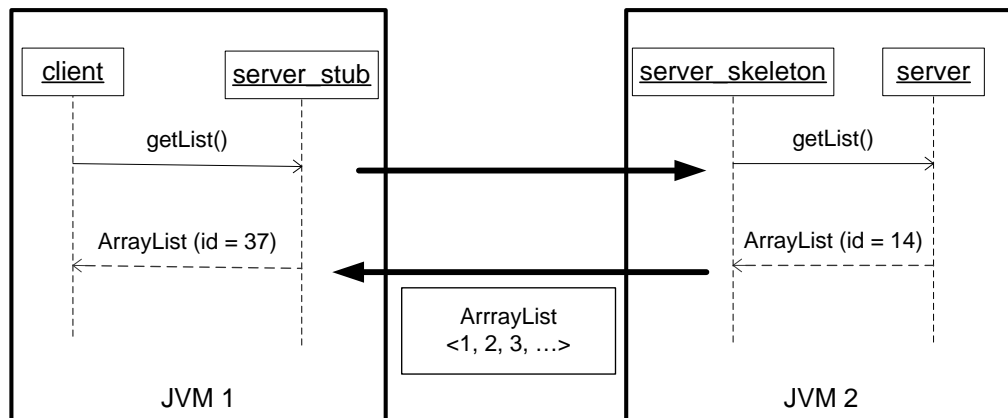
# Parameterübergabe bei Basisdatentypen

- Auf Senderseite werden Werte „gemarkshaled“ und über das Netzwerk gesendet
- Auf der Empfängerseite werden die Werte wieder entpackt und an Client geliefert



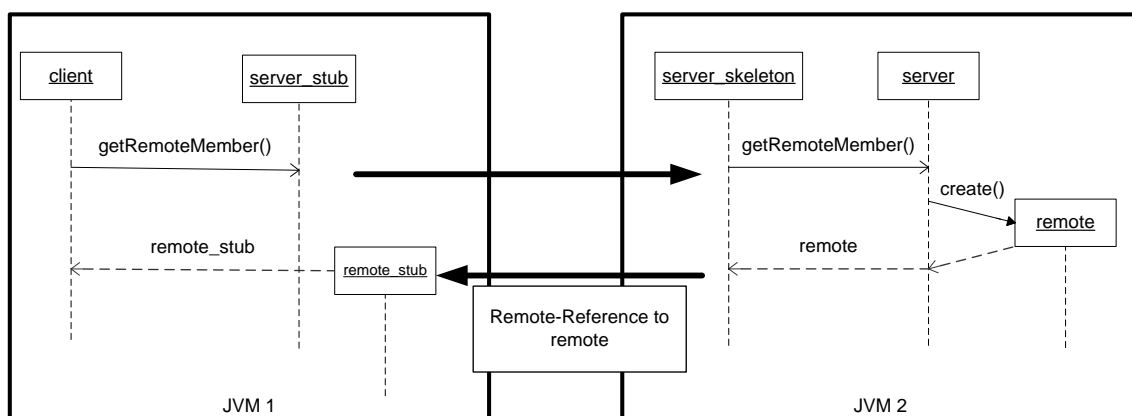
## Parameterübergabe bei serialisierbaren Objekten

- Inhalt des Objekts wird über den Java-Serialisierungsmechanismus serialisiert und über das Netzwerk übertragen
- auf der Empfängerseite wird ein neues Objekt mit gleichem Inhalt aufgebaut
- Problem, wenn Klasse des serialisierten Objekts auf Empfängerseite nicht bekannt ist!
  - siehe Nachladen von Code



## Parameterübergabe bei Remote-Objekten

- Es wird kein Wert übergeben sondern
- über das RMI-System wird eine Remote-Referenz auf das Server-Objekt übermittelt
- auf der Clientseite wird dafür ein Stub erzeugt und eine Referenz auf den Stub dem Empfänger übergeben



## Beispiel Banksystem

- BankManager ist ein Remote-Objekt, das Zugriff auf Kunden (Customer) und Konten (Accounts) bietet
- Sowohl Account als auch Customer sind Remote-Objekte
- BankManager bietet den Einstiegspunkte und wird als einziges Objekt registriert
- Vom BankManager holt sich das Client-Programm die weiteren Remote-Objekte



## Beispiel Banksystem: Interface für Remote-Objekte

```
public interface Account extends Remote {
    public BankManager getBankManager() throws RemoteException;
    public Customer getCustomer() throws RemoteException;
    public long getBalance() throws RemoteException;
    public long getCash(long amount) throws RemoteException;
}

public interface Customer extends Remote {
    public BankManager getBankManager() throws RemoteException;
    public String getName() throws RemoteException;
}

public interface BankManager extends Remote {
    public Account getAccount(String accountNumber) throws RemoteException;
    public Customer getCustomer(String clientName) throws RemoteException;
}
```

**Remote-Objekte**



## Beispiel Banksystem: BankManager Implementierung

- BankManager verwaltet Mengen von Kunden und Konten
- BankManager erlaubt Zugriff auf Konto und Kunde

```
public class BankManagerImpl implements BankManager {
    private Map<String, AccountImpl> accounts;
    private Map<String, CustomerImpl> customers;

    public BankManagerImpl() throws RemoteException {
        initialize();
    }

    public Account getAccount(String accountNumber) {
        return accounts.get(accountNumber).getRemoteStub();
    }

    public Customer getCustomer(String customerName) {
        return customers.get(customerName);
    }

    private void initialize() throws RemoteException {
        accounts = new HashMap<String, AccountImpl>();
        customers = new HashMap<String, CustomerImpl>();
        // ...
    }
}
```



## Beispiel Banksystem: Customer Implementierung

```
public class CustomerImpl extends UniCastRemoteObject
    implements Customer {

    private static final long serialVersionUID = 1L;
    private BankManager bankManager;
    private String name;

    public CustomerImpl(BankManager bankManager, String name)
        throws RemoteException {
        super();
        this.bankManager = bankManager;
        this.name = name;
    }

    public BankManager getBankManager() {
        return bankManager;
    }

    public String getName() {
        return name;
    }
}
```



## Beispiel Banksystem: Account Implementierung

```
public class AccountImpl implements Account {  
  
    private Account remoteStub;  
    private BankManager bankManager;  
    private Customer customer;  
    private int balance;  
  
    public AccountImpl (BankManager bankManager, Customer owner,  
        int startBalance) throws RemoteException {  
        this.bankManager = bankManager;  
        customer = owner;  
        balance = startBalance;  
        remoteStub = (Account)  
            UniCastRemoteObject.exportObject(this, 0);  
    }  
  
    public long getBalance() { return balance; }  
  
    public BankManager getBankManager() { return bankManager; }  
  
    public long getCash(long amount) { ... }  
  
    public Customer getCustomer() { return customer; }  
  
    public Account getRemoteStub() { return remoteStub; }  
}
```



## Beispiel Banksystem: Server-Programm

- Server-Programm legt BankManager an und registriert diesen als einziges Objekt

```
public class Server {  
  
    public static void main(String args[])  
        throws RemoteException, MalformedURLException {  
  
        BankManagerImpl bm = new BankManagerImpl ();  
        RemoteStub bmr = UniCastRemoteObject.exportObject(bm);  
        Naming.rebind("//localhost/BankSystem", bmr);  
    }  
}
```



## Beispiel Banksystem: Client-Programm

- Client-Programm holt sich über RMIRegistry den BankManager
- greift damit auf Account-Objekte und Customer-Objekte zu

```
public class Client {  
    public static void main(String[] args) throws  
        MalformedURLException, RemoteException, NotBoundException {  
        BankManager bm = (BankManager) Naming  
            .lookup("rmi://localhost:1099/BankSystem");  
  
        Account account = bm.getAccount("4461");  
        Customer customer = account.getCustomer();  
        String name = customer.getName();  
        Long cash = account.getBalance();  
  
        NumberFormat currencyFormat = NumberFormat  
            .getCurrencyInstance(Locale.US);  
        String balanceString = currencyFormat.format(cash);  
        System.out.println(name + "'s account has " +  
            balanceString);  
    }  
}
```



## equals und == bei Remote-Objekten

- Bei jedem Zugriff auf ein Remote-Objekt wird am Client ein neues Stub-Objekt angelegt  
→ (stub1 != stub2) für zwei Client-Stubs für selbes Remote-Objekt

```
Account stub1 = bm.getAccount("4461");  
Account stub2 = bm.getAccount("4461");
```

```
if (stub1 == stub2)
```

false

- Client-Stubs für selbes Remote-Objekt sind equals  
→ stub1.equals(stub2) für zwei Client-Stubs für selbes Remote-Objekt

```
Account stub1 = bm.getAccount("4461");  
Account stub2 = bm.getAccount("4461");
```

```
if (stub1.equals(stub2))
```

true

- Implementiert Remote-Objekt aber equals, wird dieses nicht berücksichtigt  
→ Überschriebenes equals kann am Client nicht verwendet werden

Beispiel nächste Seite



## Beispiel: equals bei Remote-Objekten [1/2]

### Remote-Objekte Car

```
public interface Car extends Remote {
    public int getNr() throws RemoteException;
}
```

CarImpl definiert equals, wenn Objekte gleichen Wert nr

```
public class CarImpl extends UnicastRemoteObject implements Car {
    private int nr;

    protected CarImpl(int nr) throws RemoteException {
        super();
        this.nr = nr;
    }
    ...

    public boolean equals(Object obj) {
        if (obj instanceof CarImpl) {
            CarImpl c = (CarImpl) obj;
            return nr == c.nr;
        }
        return false;
    }
}
```



## Beispiel: equals bei Remote-Objekten [2/2]

### Server CarStore erzeugt immer neue CarImpl -Objekte

```
public interface CarStore extends Remote {
    public Car getCar(int nr) throws RemoteException;
}
```

```
public class CarStoreImpl extends UnicastRemoteObject implements CarStore {
    public CarStoreImpl() throws RemoteException {
        super();
    }

    @Override
    public Car getCar(int nr) throws RemoteException {
        return new CarImpl(nr);
    }
}
```

Diese Objekte werden am Client nie als equals erkannt

```
public class Client {

    public static void main(String[] args) throws RemoteException ... {
        Registry reg = LocateRegistry.getRegistry(2222);
        CarStore cs = (CarStore) reg.lookup("CarStore");

        Car c1a = cs.getCar(1);
        Car c1b = cs.getCar(1);
        if (c1a.equals(c1b))
```

false



# Remote-Objekte

Motivation

Grundarchitektur

Implementierung von Remote-Objekten

Parameterübergabe

Callbacks

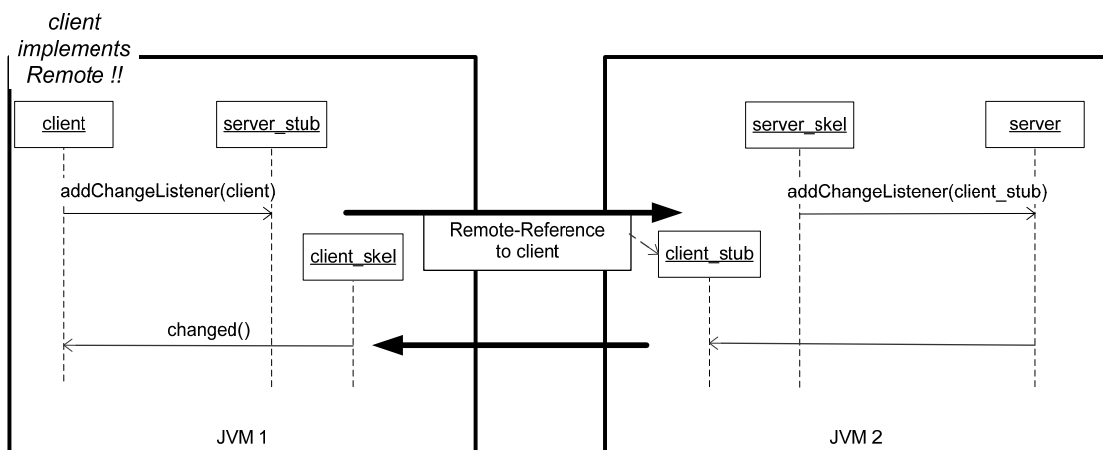
Distributed Garbage Collection

Verteilung und Nachladen von Code



# Callbacks

- In machen Fällen ist es notwendig, dass der Server dem Client eine Meldung schickt.
  - z.B. Benachrichtigung über Änderungen
- In diesem Fall sind die Rolle von Client und Server zu vertauschen
  - am Client muss ein Remote-Objekt existieren
  - eine Remote-Referenz wird dem Server übergeben
  - am Server wird ein Stub für das Remote-Client-Objekt angelegt



# Beispiel Banksystem: Remote Change Notification

- Remote-Listener-Interface für Client

```
public interface RemoteAccountListener extends Remote {  
    public void accountChanged(RemoteAccountEvent e) throws RemoteException;  
}
```

- BankManager wird um Remote-Methode addAccountListener erweitert

```
public interface BankManager extends Remote {  
    public void addRemoteAccountListener(RemoteAccountListener l) throws RemoteException;  
}  
  
public class BankManagerImpl implements BankManager {  
    ...  
    public void addRemoteAccountListener(RemoteAccountListener l) throws RemoteException {  
        listeners.add(l);  
    }  
}
```

- Der Client muss RemoteAccountListener implementieren und sich als Listener bei der Bank anmelden;

```
public class BankUser extends UniCastRemoteObject implements RemoteAccountListener {  
    public BankUser(BankManager bm) throws RemoteException {  
        super();  
        this.bm = bm;  
        // add client as account listener at the bank manager  
        bm.addRemoteAccountListener(this);  
    }  
  
    public void accountChanged(RemoteAccountEvent e) throws RemoteException { ... }  
}
```



# Remote-Objekte

Motivation

Grundarchitektur

Implementierung von Remote-Objekten

Parameterübergabe

Callbacks

Distributed Garbage Collection

Verteilung und Nachladen von Code



# Distributed Garbage Collection

- Garbage Collector gibt Objekte frei, wenn sie nicht mehr referenziert werden
- Bei Remote-Objekten ist es dem Garbage Collector aber nicht möglich zu entscheiden, ob Remote-Clients das Objekt noch referenzieren

→ Distributed Garbage Collector arbeitet mit:

- *Reference Counting*: Es werden die Zugriffe von Remote-Clients auf die Remote-Objekte gezählt
- *Lease Time*: Greift ein Client eine bestimmte Zeit – der Lease Time – nicht auf das Remote-Objekt zu, wird angenommen, dass der Client das Objekt nicht mehr benötigt.

Konsequenz: Beim Client ist es möglich, dass Remote-Objekte verschwinden und er mit ungültigen Remote-Referenzen umgehen können muss.

Lease Time:

- System Property `java.rmi.dgc.leaseValue`
- Standardwert 10 min
- Einstellung als Option bei java: `java -Djava.rmi.dgc.leaseValue=20000`



# Remote-Objekte

Motivation

Grundarchitektur

Implementierung von Remote-Objekten

Parameterübergabe

Callbacks

Distributed Garbage Collection

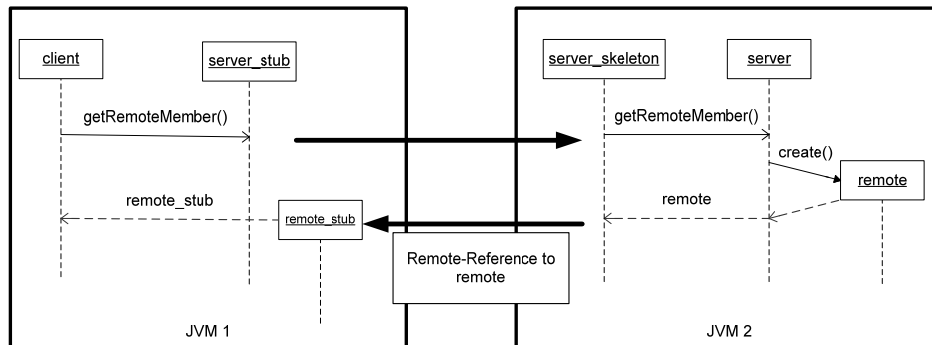
Verteilung und Nachladen von Code



# Verteilung und Nachladen von Klassen

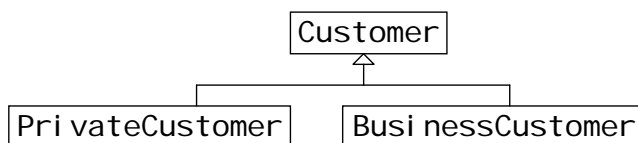
- Bei Zugriff auf Remote-Objekte ist es nötig, dass Code vom Server nachgeladen wird
  - für Stubs
  - für Parameter und Rückgabewerte

## Beispiel: Stub



# Beispiel Nachladen von Klassen

```
public interface BankManager extends Remote {
    public Customer getCustomer(String clientName) throws RemoteException;
}
```



Es existieren Spezialisierungen von Customer! Diese sind Client eventuell nicht bekannt!

```
public class BankManagerImpl extends UniCastRemoteObject implements BankManager {
    public Customer getCustomer(String customerName) throws RemoteException {
        ...
        return busi nessCl i ents. get(customerName);
    }
}
```

Client: Müsste alle Spezialisierungen von Customer kennen

Konkretes Objekt vom Typ **PrivateCustomer\_Stub** oder **BusinessCustomer\_Stub**

```
try {
    Customer customer = account.getCustomer();
} catch (RemoteException remoteException) {
    System.err.println(remoteException);
}
```



## Nachladen von Code

- Um Code von einer externen Site nachladen zu können, ist folgendes zu tun
  - es muss beim Client ein RMI SecurityManager installiert sein (sonst kann kein externer Code geladen werden)
  - Es müssen Permissions für den Client gesetzt werden:
    - Socketverbindung zum Server für RMI über Port 1099>
    - Socketverbindung zum Nachladen von Code hier über Webserver, d.h. Port 80 (oder 8080)

### Beispiel:

Clientprogramm:

```
public class MyClient {
    public static void main(String[] args) {
        System.setSecurityManager(new RMI SecurityManager());
        System.setProperty("java.security.policy", "client.policy");
        ...
    }
}
```

Policy-Datei für Client:

```
grant {
    permission java.net.SocketPermission "server-url: 1024-65535", "connect";
    permission java.net.SocketPermission "server-url: 80", "connect";
    permission java.net.SocketPermission "server-url: 8080", "connect";
}
```



## Verteilung des Klassencodes

- Die Klassen sollen zur Verteilung auf 3 Teile aufgeteilt werden:

### server:

- alle Klassendateien, die für die Ausführung des Servers erforderlich sind
- alle Stub-Klassen

### download:

- alle Klassendateien, die dynamisch vom Client nachgeladen werden sollen
- inklusive alle abhängigen (z.B. Basisklassen und Interfaces)

### client:

- alle Klassendateien, die unmittelbar vom Client verwendet werden
- die Policy-Datei (z.B. client.policy)

- Die 3 Teile werden dann folgend verteilt:

### server:

- auf dem Rechner des Servers

### download:

- bei Verwendung eines Webserver ins download-Verzeichnis des Webserver

### client:

- auf den Rechnern des Client



## Starten der verteilten Programme

- Am Server

- Starten des RMIRegistry-Programms

```
MI crosoft WI ndows XP  
> rmi regi stry
```

- Starten des Server-Programms mit Angabe des Download-Verzeichnisses als Codebase

```
> j ava -Dj ava. rmi . server. codebase=http: //l ocal host/downl oad/  
MyServerApp
```

- Am Client

- Starten des Client-Programms, hier zusätzlich unter Angabe einer Policy-Datei

```
> j ava -Dj ava. securi ty. pol i cy=cl i ent. pol i cy MyCl i entApp
```



## Literatur

- Horstmann, Cornell, Core Java 2, Band 2 Expertenwissen, Markt und Technik, 2002: Kapitel 5
- Krüger, Handbuch der Java-Programmierung, 3. Auflage, Addison-Wesley, 2003, <http://www.javabuch.de>: Kapitel 46
- Fundamentals of RMI, Short Course, <http://java.sun.com/developer/onlineTraining/rmi/>

