Fachhochschule München
Fachbereich 07 Informatik

Anforderungen objektorientierter Sprachen an die Datenkommunikation

am Beispiel von Java

WS 2003

Von

Klaus Baumgartner

und

Matthias Nau

Ausarbeitung unter
http://www.illusioni.de/klaus/fh/dako/seminararbeit.pdf

Inhalt

1. Definierte Schnittstellen 3
2. Definiertes Fehlerverhalten 4
3. Objektserialisierung 5
4. Methodenaufrufe über Plattformgrenzen hinweg 8
5. Weiterführende Informationen 13
6. Fragen 13

1. Definierte Schnittstellen

In allen aktuellen objektorientierten Programmiersparchen werden sämtliche Funktionen, die für die Kommunikation zwischen Prozessen oder Systemen notwendig sind, in Form von Bibliotheken oder Programmierschnittstellen (APIs) bereitgestellt.

Hauptaugenmerk ist dabei die Transparenz für den Benutzer. Programmierer erwarten von IO-APIs ein Blackboxverhalten. Die Schnittstellen und die dahinterliegende Semantik sind dokumentiert. Die Umsetzung (Implementierung) ist nicht von Interesse.

In Java benutzt der Programmierer je nach Bedarf verschiedene Teile der Core-API, einer auf jeder Java-Plattform vorhandenen Menge von Programmierschnittstellen. Die API ist dabei untergliedert in eine Reihe von Packages. Die für die Kommunikation zuständigen Packages sind u.a. java.net, java.io oder java.rmi und deren Subpackages.

Natürlich findet Kommunikation nicht nur zwischen Programmen untereinander statt. Kommunikationsschnittstellen zu Datenbanksystemen, Object Request Brokers (ORBs) oder anderen mittlerweile standardisierten Systemen werden hier jedoch nicht behandelt. Wir konzentrieren uns auf die Basis der Kommunikation, auf der auch Implementierungen für oben genannte Bereiche aufsetzen.

Eine Kommunikations-API kann im ISO-7-Schicht Modell in den Schichten 5 und 6 angesiedelt werden. Aufgaben dieser Schichten sind die Kommunikationssteuerung und die Darstellung von Daten. Wichtigste Funktion der Kommunikationssteuerung (Ebene 5) ist die Wahrung eines Kontextes (Session). Auf Ebene 6 liegt die Verantwortung dafür, wie binäre Informationen zu formatieren bzw. zu interpretieren sind.

Bei Kommunikation über TCP/IP ist es üblich, Objekte zu definieren, die den Kontext reprästentieren. Der Typ für diese Objekte heißt Socket.

Für die Kommunikation zwischen zwei Partnern müssen die Partner über Sockets eine Verbindung hergestellt haben. Jeder Partner kann von seinem Socket Objekt nun für das Senden bzw. Empfangen von Daten je ein Stream beziehen.

Wie Zeichen für die Übertragung kodiert bzw. dekodiert werden, ist teilweise in der Programmiersprache definiert. Andere Informationen wiederum müssen der API vom Programm mitgeteilt werden können. Hierzu gibt es keine allgemeine, in den verschiedenen Sprachen gleiche, Vorgehensweise. In Java gibt es Schnittstellen für Codepages u.a. mit Implementierungen für Standardzeichensätze (z.B. ASCI, EBCDIC, UNICODE) und der Möglichkeit, das Encoding bzw. Decoding für eigene oder weniger verbreitete Zeichensätze selbst zu implementieren.

Zu Sockets und Streams folgt später ein Beispiel.

2. Definiertes Fehlerverhalten

Bei jeglicher Form von Kommunikation können im alltäglichen Betrieb Fehlersituationen entstehen. Beispiele wären Verbindungsfehler, Timeoutüberschreitungen oder Pufferüberläufe. Jeder OO-Sprache verfügt über eine standardisierte Vorgehensweise für den Umgang mit Ausnahmesituationen oder Fehlern. Fehler in der Kommunikation werden auch über diese Mechanismen behandelt.

In Java oder C++ wird Code, der zu Fehlern führen kann, in einen sogenannten try Block gesetzt. Einer oder mehrere catch-Blöcke beinhalten den Code, der für die Behandlung eines (typisierten) Fehlers notwendig ist.

Ein kurzes Beispiel soll zeigen, wie Zeichenketten in Java zwischen zwei Plattformen (zwei unterschiedliche Virtuelle Maschinen (VMs) oder Rechner) übertragen werden können.

```
public class Sender {

public static final int PORT = 12345;

private Socket so;
private BufferedWriter bWriter;

  public Sender(String host) throws IOException {
    so = new Socket(host, PORT); //1
    OutputStream os = so.getOutputStream();
    bWriter = new BufferedWriter(new OutputStreamWriter(os));
  }

  public void send(String s) throws IOException {
    bWriter.write(s);
    bWriter.newLine();
    bWriter.flush();
  }

  public void close() throws IOException {
    bWriter.close();
    so.close();
  }

  public static void main(String[] args) { //2
    try {
      Sender sender = new Sender("192.168.0.2"); //3
      for (int i = 0; i < 10; i++) {
        sender.send("Hello World");
      }
      sender.close();
    }
    catch (IOException ex) {
      System.out.println("Fehler");
      ex.printStackTrace(System.out);
    }
  }
}
```

Die Klasse Sender benutzt ein Socket-Objekt, das mit der IP-Adresse des Empfängers und einem Port initialisiert wird (Zeile //1).

In der Methode main() (Zeile //2) ist gut das Java- und C++ typische try{}-catch(){} Konstrukt zum Abfangen von Fehlern zu erkennen.

Die IP-Adresse, die in Zeile //3 an den Konstruktor von Sender übergeben wird, ist hier fest gecoded. In der Praxis würde man dieses Argument natürlich parametrisieren.

```
public class Receiver {
  public static final int PORT = 12345;
  private Socket socket;
  private BufferedReader bReader;

  public Receiver() throws IOException {
    ServerSocket sSocket = new ServerSocket(PORT);
    socket = sSocket.accept();
    InputStream iStream = socket.getInputStream();
    bReader = new BufferedReader(new InputStreamReader(iStream));
  }

  public String receive() throws IOException {
    String res = bReader.readLine();
    return res;
  }

  public void close() throws IOException {
    bReader.close();
    socket.close();
  }

  protected void finalize() {
    try {
      close();
    }
    catch (Throwable t) {}
  }

  public static void main(String[] args) {
    System.out.println("Starting receiver");
    try {
      Receiver r = new Receiver();
      boolean run = true;
      while (run) {
        String msg = r.receive();
        if (msg == null) {
          run = false;
        }
        else {
          System.out.println(msg);
        }
      }
      System.out.println("No more messages.");
    }
    catch (IOException ioe) {
      System.out.println("IOException! No message received.");
    }
  }
}
```

3. Objektserialisierung

Objektserialisierung ist die Technologie, die eingesetzt wird, um Objekte mit ihren aktuellen Attributen in Bytefolgen umzuwandeln (respektive Bytefolgen ind gültige Objekte umzusetzen).

Ziel der Objektserialisierung ist die sogenannte Persistenz - das Erhalten der Zustände von Objekten über die Ausführungszeit eines Programms bzw. die Grenzen des Systems hinaus.

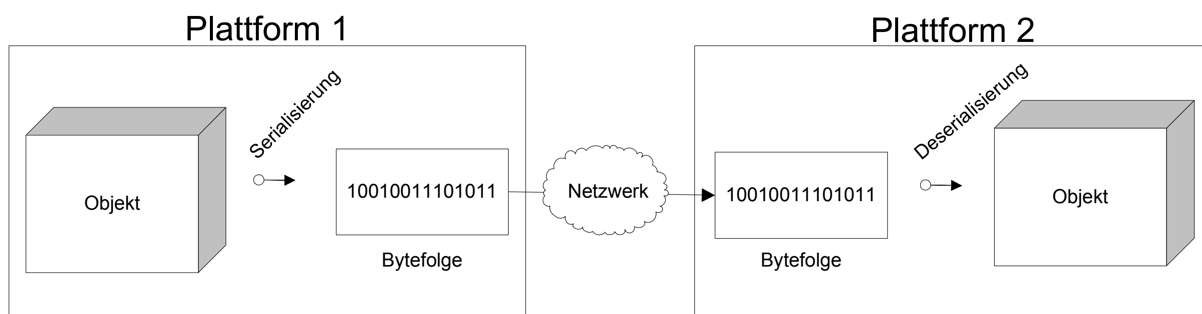
Wichtig dabei ist eine definierte Serialisierungssemantik. Objekte können über zwei Arten in Bytefolgen übersetzt werden.

Die Erste ist das Implementieren des Interfaces java.io.Serializable. Instanzen von Klassen, die dieses Interface implementieren, werden serialisiert, indem alle Felder der Instanz nacheinander in Bytes umgewandelt werden. Für primitive Datentypen gibt es festgelegte (plattformübergreifende) Binärformate. Referenzierte Objekte werden rekursiv serialisiert.

Die zweite Art, Objekte zu serialisieren, ist das Implementieren des Interfaces java.io.Externalizable. Dieses Interface schreibt zwei Methoden vor, in denen der Programmierer ein Objekt übergeben bekommt, in das er Bytes schreiben, bzw. aus dem er Bytes lesen kann. Dadurch kann er selbst die Serialisierung programmieren und damit eine gewünschte Semantik erzielen.

Da sich Objekte nur innerhalb eines Systems gegenseitig referenzieren können, ist es notwendig, zu unterscheiden, ob beim Schreiben eines Objektes referenzierte Objekte ebenfalls serialisiert werden sollen. In Java bewirkt das Schlüsselwort transient bei Feldern, dass diese bei der Serialisierung nicht berücksichtigt werden.

Ein weiteres Beispiel zeigt, wie komplexe Objekte (hier von einem eigenen Datentyp Person) über ein Netzwerk übertragen werden können.



Das Programm ist ähnlich aufgebaut wie das vorherige Sender/Receiver Beispiel. Jetzt werden allerdings statt Zeichenketten Objekte übertragen, die auf Plattform 1 (Sender-Plattform) zunächst in 1/0 Folgen konvertiert werden müssen, welche nach der Übertragung auf Plattform 2 (Receiver-Plattform) „deserialisiert“, also wieder zu gültigen Objekten auf dem Heap umgewandelt werden müssen.

```
package dako;

import java.net.*;
import java.io.*;


public class ObjectSender {

public static final int PORT = 12345;

private Socket so;
private ObjectOutputStream oOS;

  public ObjectSender(String host) throws IOException {
    so = new Socket(host, PORT);
    OutputStream os = so.getOutputStream();
    oOS = new  ObjectOutputStream(os);
  }

  public void send(Object o) throws IOException {
    oOS.writeObject(o); //1
    oOS.flush();
  }

  public void close() throws IOException {
    oOS.close();
    so.close();
  }

  public static void main(String[] args) {
    try {
      ObjectSender sender = new ObjectSender("localhost");
      Person p = new Person("Hans", 38);
      Person v = new Person("Herbert", 67);
      Person m = new Person("Hannelore", 59);
      p.setParents(v, m);
      sender.send(p);
      sender.close();
    }
    catch (IOException ex) {
      System.out.println("Fehler");
      ex.printStackTrace(System.out);
    }
  }
}
```

Objekte, die an die writeObject() Methode eines ObjectOutputStreams übergeben werden, müssen serialisierbar sein (Zeile //1).

Die in unserem Beispiel verwendete Klasse Person implementiert das Flag-Interface „Serializable“ und verlässt sich auf den ObjectOutputStream, der mittels der Reflection-API erkennt, dass Instanzen der Klasse die Felder _name und _alter besitzen. Das Schlüsselwort „transient“ vor den Feldern _vater und _mutter signalisiert dem ObjectOutputStream, dass die referenzierten Objekte nicht rekursiv mitserialisiert werden sollen.

```
public class Person implements Serializable {

  private String  name;
  private int _alter;

  private transient Person  vater,  mutter;

  public Person(String name, int alter) {
     name = name;
    _alter = alter;
  }

  public String getName() {
    return  name;
  }

  public int getAlter() {
    return _alter;
  }

  public Person getVater() {
    return _vater;
  }

  public Person getMutter() {
    return  mutter;
  }

  public void setParents(Person v, Person m) {
    _vater = v;
    _mutter = m;
  }

  public String toString() {
   return "Person[" +  name + ", " +  alter + "]";
  }

}
```

4. Methodenaufrufe über Plattformgrenzen hinweg

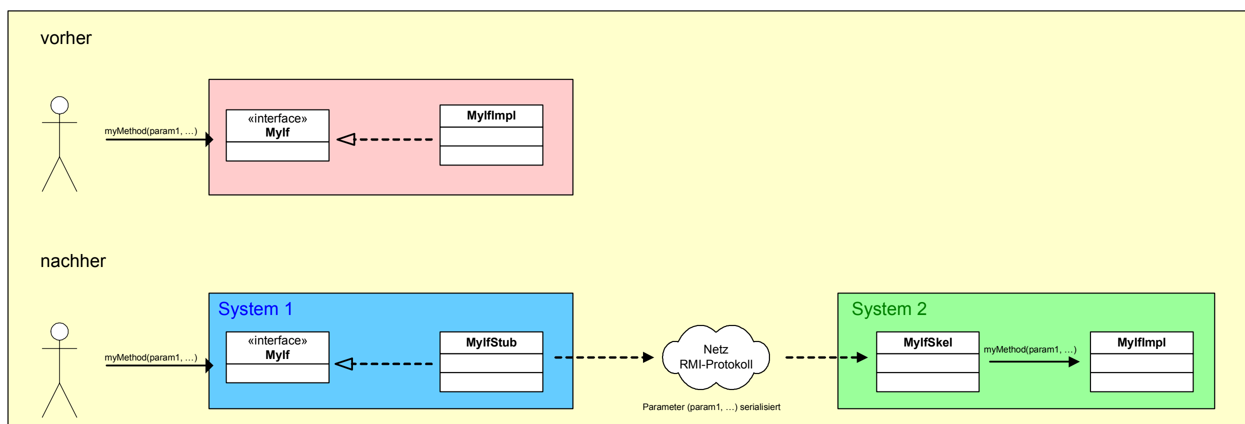
Verwendet man ausschließlich Sockets und Streams, um zwischen zwei Softwarekomponenten zu kommunizieren, so entsteht zwangsläufig bei steigender Komplexität die Notwendigkeit, ein eigenes Kommunikationsprotokoll zu definieren. Oft handelt es sich bei den Daten lediglich um Informationen, welche Funktionen auf dem entfernten System ausgeführt werden sollen oder um den Austausch von Parametern und Rückgabewerten. Die Umsetzung von Hand ist langwierig und außerdem fehlerträchtig.

In Java gibt es eine Technologie, die den Programmierer bei der Entwicklung von Remote-Funktionsaufrufen unterstützt: RMI steht für Remote Method Invocation und besteht aus einem Protokoll für die Übertragung von Funktionsaufrufen, Parametern und Rückgabewerten, sowie einem Tool (dem RMI-Compiler rmic), das aus vorhandenen Klassen und deren zugehörigen Schnittstellendefinitionen sogenannte Stubs und Skeletons generiert. Stubs und Skeletons führen die Kommunikaiton zwischen dem Rechner, auf dem die Methode aufgerufen wird (Client), und dem Rechner, auf dem sie ausgeführt wird (Server), durch. Ein Stub ist ein Objekt, das die Schnittstelle der Klasse implementiert und auf dem Client Aufrufe an die Methoden

entgegennimmt, um sie an ein Skeleton weiterzuleiten, welches auf dem Server liegt und die Anfragen in Methodenaufrufe auf die eigentlich implementierende Klasse umsetzt.

Da Parameter u. Rückgabewerte vom Stub zum Skeleton oder umgekehrt über das Netzwerk gesendet werden müssen, müssen sie primitiv oder serialisierbar sein. Der Versuch, nicht-serialisierbare Objekte an Remote-Methoden zu übergeben, führt zu einer Exception.

Die folgende Grafik zeigt ein Programm vor und nach der Erweiterung um einen Remote-Zugriff.



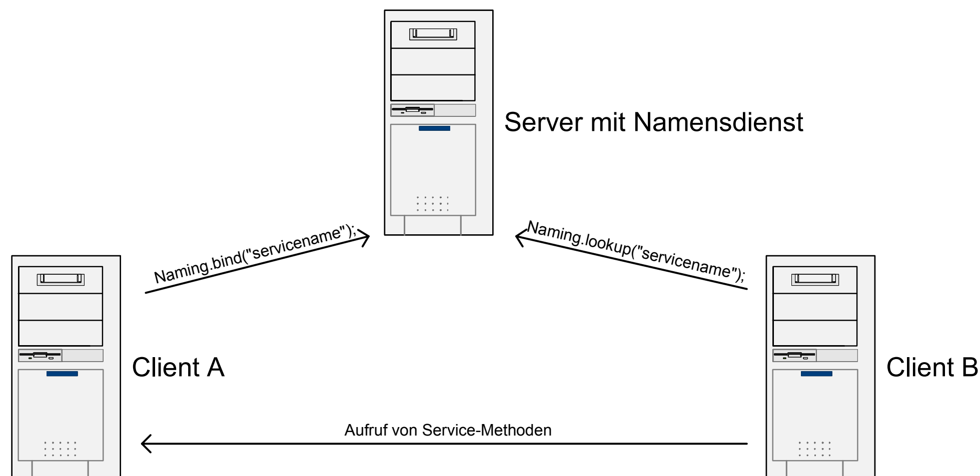
Ein System vor und nach der Aufteilung auf Client und Server (Two-tier Architektur)

Die Vorgehensweise beim Erstellen von Programmen mit Remote-Aufrufen sei kurz in Stichpunkten beschrieben:

1. Definieren einer Schnittstelle (Remote-Interface) und einer Implementierungsklasse
2. Aufruf der RMI-Compilers. Dieser erzeugt Stub und Skeleton.
3. Servercode zum Registrieren der Implementierung bei einem Namensdienst
4. Clientcode zum Beziehen des Stubs von einem Namensdienst
5. Aufruf der Stub-Methoden. Der Stub kommuniziert über das RMI-Protokoll mit dem Skeleton, welches wiederum die Methoden der Implementierungsklasse aufruft.

Damit ablaufende Programme nach benannten Services fragen können, gibt es einen Namensdienst, der symbolische Servicenamen auf Implementierungsanbieter abbildet. Ein Implementierungsanbieter ist eine Laufzeitumgebung mit einer instanziierten Klasse, die das Service-Interface implementiert. Der besagte Namensdienst kann Teil eines Application-Containers sein. Im einfachsten Fall ist es die „RMIRegistry“ – ein mit dem JDK ausgeliefertes Programm, das auf einem festen Server gestartet werden kann, welcher jedem Client bekannt ist.

Die folgende Grafik zeigt ein einfaches Setup das aus zwei Clients und einem Server mit bekannter, fester IP-Adresse besteht, auf dem der Namensdienst abläuft.



Zunächst registriert Client A seine Service-Implementierung beim Namensdienst unter einer selbst festgelegten Kennung. Später fragt Client B mit der Methode „lookup()“ den Namensdienst nach der registrierten Service-Implementierung, bekommt den Stub übertragen und kann auf diesen die Service-Methoden aufrufen.

Wichtig: Der Begriff „Client“ bezieht sich hier nur auf den Namensdienst. Client A ist später die Plattform, auf der die Service-Methoden ausgeführt werden (Server im RMI-Kontext).

Abstrahiert man ein wenig, kann man diese Architektur der einer Netzwerkumgebung mit Object-Brokerage gegenüberstellen. Mit CORBA gibt es hier auch eine Technologie, die sprachunabhängig ist und einheitliche Dateninterpretation definiert.

In einem letzten Beispiel - einer kleinen Applikation, die Auskünfte über Personen anhand ihrer Namen erteilt - soll der Nutzen der oben beschriebenen Architektur veranschaulicht werden.

Auf dem Server wird (in der Methode main() der Klasse RMIService) eine Instanz der Klasse PersonLookupImpl erzeugt und beim Namensservice registriert. Das Interface PersonLookup beschreibt die aufrufbaren Methoden. Die Klasse RMIClient enthält den Code, der auf einem beliebigen Rechner im Netzwerk ausgeführt werden kann. Alle weiteren Klassen sind vom RMI-Compiler produzierter Bytecode.

```
Import java.rmi.*;

public interface PersonLookup extends Remote {
  public int getAge(String name) throws RemoteException;
  public Person getFather(String name) throws RemoteException;
  public Person getMother(String name) throws RemoteException;
}
```

Das Inteface PersonLookup erweitert das Interface java.rmi.Remote und signalisiert dem RMICompiler damit, dass für dieses Interface (und seine Implementierung) ein Stub und ein Skeleton erzeugt werden sollen.

```
public class PersonLookupImpl
    implements PersonLookup, Serializable {
  private final Map pMap;

  public PersonLookupImpl(Map pMap) {
    this.pMap = pMap;
  }

  public int getAge(String name) {
    Person p = resolveName(name);
    int res = p.getAlter();
    return res;
  }

  public Person getFather(String name) {
    Person p = resolveName(name);
    Person res = p.getVater();
    return res;
  }

  public Person getMother(String name) {
    Person p = resolveName(name);
    Person res = p.getMutter();
    return res;
  }

  private Person resolveName(String name) {
    Person p = (Person) pMap.get(name);
    return p;
  }
}
```

PersonLookupImpl ist eine einfache Beispiel-Implementierung des Interfaces PersonLookup. Die genaue Umsetzung spielt für unser Beispiel keine große Rolle. Der Code ist nur der Vollständigkeit halber (und für besseres Verständnis) angegeben.

```
public class RMIService {
  public static final int PORT = 2299;

  public static void main(String[] args) {
    try {
      System.out.println("Initializing data");
      Map pMap = initPersons();
      PersonLookupImpl pImpl = new PersonLookupImpl(pMap);
      System.out.println("Registering service");
      Naming.rebind("rmi://localhost:" + PORT + "/personlookup", pImpl);//1
      System.out.println("OK");
    }
    catch (Throwable t) {
      System.out.println("An error occured:");
      t.printStackTrace();
    }
  }

//... (Fortsetzung nächste Seite)
```

```
//... (Fortsetzung)

  private static Map initPersons() {
    Map pMap = new HashMap(); //maps names to Person objects
    Person hermann = add(pMap, "Hermann", 23);
    Person hugo = add(pMap, "Hugo", 5);
    Person hermine = add(pMap, "Hermine", 42);
    hugo.setParents(hermann, hermine);
    return pMap;
  }

  private static Person add(Map m, String name, int age) {
    Person p = new Person(name, age);
    m.put(name, p);
    return p;
  }
}
```

Die Klasse RMIService benutzt die Methode „rebind()“ (Zeile //1), um die Service-Implementierung beim Namensdienst auf dem angegebenen Rechner zu registrieren. Wir benutzen im Code die Methode „rebind()“ anstelle von „bind()“, da sie sich fast gleich verhält, aber im Fall, dass unter dem angegebenen Service-Namen bereits ein Dienst registriert ist, keine Ausnahme wirft, sondern die vorhandene Dienstimplementierung durch unsere ersetzt.

```
public class RMIClient {
  public static final int PORT = 2299;
  public static final String hostname = "magdalena.illusioni.din";

  public static void main(String[] args) {
    try {
      PersonLookup lookup = (PersonLookup) Naming.lookup("rmi://" +
      hostname + ":" + PORT + "/personlookup");

      int ageHermine = lookup.getAge("Hermine"); //2
      System.out.println("Hermine is " + ageHermine + " years old.");
      Person father = lookup.getFather("Hugo");
      System.out.println("Hugo's father is " + father);
    }
    catch (NotBoundException nbe) { //3
      System.out.println("Service not available.");
    }
    catch (Throwable t) {
      System.out.println("Unexpected Exception:");
      t.printStackTrace();
    }

  }
}
```

RMIClient ruft die Methode „lookup()“ (Zeile //2) auf, um die Dienstimplementierung zu erfragen.

Ist zu dem symbolischen Namen keine Implementierung registriert, so wird dies durch eine NotBoundException (Zeile //3) signalisiert.

Um das Beispiel auszuführen, sind noch folgende Schritte auszuführen:

1. Kompilieren der oben aufgeführten Klassen
2. Aufruf des RMICompilers: „rmic dako.rmi.PersonLookup“
3. Starten des Namensdienstes: „rmiregistry“
4. Starten des Programms RMIService
5. Starten des Clients

5. Weiterführende Informationen

Für weitere Informationen zum Thema RMI empfehlen wir die Internet-Seite http//www.developer.java.sun.com/. Fragen zu dieser Ausarbeitung bzw. zum Vortrag beantworten wir gerne – mailto:klaus@illusioni.de. ;o)

6. Fragen

1. *Welche Informationen benötigt man in einem Programm, um mittels eines Socket eine TCP/IP-Verbindung aufzubauen?*

   IP-Adresse und Port

2. *Wo liegt der Unterschied zwischen den beiden Interfaces Serializable und Externalizable zum Serialisieren von Java-Objekten?*

   Instanzen von Klassen, die das Interface Serializable implementieren, werden von der Java-VM in Bytefolgen umgewandelt. Für primitive Datentypen gibt es festgelegte Binärformate. Referenzierte Objekte werden rekursiv serialisiert.

   Bei Instanzen von Klassen, die das Interface Externalizable implementieren, wird vom Programmierer festgelegt, wie ein Objekt serialisiert wird. Damit kann eine gewünschte Semantik erzielt werden.

3. *Welche Bedingung müssen Parameter und Rückgabewerte von Remote-methoden erfüllen?*

   Parameter und Rückgabewerte von Remotemethoden müssen serialisierbar sein.