# Java in verteilten Systemen

Jörn Hameister, E-Mail: hameiste@rbg.informatik.tu-darmstadt.de
Informatik
Technische Universität
Darmstadt

## Abstract

*In diesem Artikel sollen die Konzepte und Schnittstellen von Java erklärt werden. Das Hauptaugenmerk liegt in der Anwendung von Java in verteilten Systemen. Es soll insbesondere die Kommunikation zwischen Client und Server erklärt werden. Dieser Artikel ist keine Einführung in das Programmieren mit Java, sondern gibt einen Einblick in die Erweiterungen von Java, die bei verteilten Systemen zum Einsatz kommen. Außerdem werden nur einige wichtige Konzepte erklärt und nicht alle, die sich in verteilten Systemen einsetzen lassen.*

## 1. Einleitung

Dieser Artikel behandelt die Möglichkeiten von Java in verteilten Systemen. In Abschnitt 2 werden verschiedene Konzepte von Java dargelegt, die für den Einsatz in verteilten Systemen geeignet sind. Applikationen und Applets werden in Abschnitt 2.1 erklärt. In Abschnitt 2.2 werden die Möglichkeiten von JDBC erläutert. In Abschnitt 2.3 wird der Umgang mit *Remote Method Invocation* (RMI) erklärt. In ihm wird genau darauf eingegangen, wie mit RMI auf entfernte Objekte zugegriffen werden kann und aus welchen Teilen eine Applikation oder ein Applet besteht, das RMI benutzt. Außerdem wird beschrieben, wie die Kommunikation zwischen Client und Server funktioniert. Dieser Abschnitt ist länger als die anderen Abschnitte, weil RMI eine der Kernkomponenten für den Einsatz von Java in verteilten Systemen ist. In Abschnitt 2.4 wird *Interface Definition Language* (IDL) und *Common Object Request Broker Architecture* (CORBA) angesprochen. Dieser Abschnitt behandelt IDL nur oberflächlich, weil IDL in Verbindung mit CORBA sehr umfangreich ist.Abschnitt 2.5 erklärt, was unter *Servlets* zu verstehen ist und wie sie funktionieren. Die Abschnitte 2.6 und 2.7 können nur die “groben” Funktionsweisen von *Java Naming and Directory Interface* (JNDI) und *Enterprise JavaBeans* (EJB) skizzieren.
In Abschnitt 3 werden Java Servlets und Applets miteinander verglichen. Es wird darauf eingegangen, worin die Unterschiede liegen und wann Applets und wann Servlets eingesetzt werden sollten.
In Abschnitt 4 soll an einem kleinen Beispiel demonstriert werden, wie die einzelnen Java-Komponenten eingesetzt werden können, um eine Anwendung für eine Internetbuchhandlung zu entwerfen.
Im letzten Abschnitt wird ein Fazit gezogen und gesagt, wie die Möglichkeiten von Java in verteilten Systemen zu bewerten sind und wie die Zukunft von Java aussehen könnte.
Die in diesem Artikel angesprochenen Komponenten werden in der Literatur unter dem Begriff “Java Enterprise“ zusammengefaßt [1]. Angemerkt sei noch, daß keinesfalls alle Aspekte von Java in verteilten Systemen angesprochen werden können. So wurde beispielsweise Jini, Java Message Service (JMS), Java Transaction API (JTA) ausgeklammert.

## 2. Konzepte und Schnittstellen von Java

In diesem Abschnitt werden die verschiedenen Konzepte und Schnittstellen von Java erklärt. Sie werden in Bezug auf verteilte Systeme untersucht.

### 2.1. Applikationen und Applets

Applikationen sind Programme, deren Code normalerweise auf einem Client ausgeführt wird. Eine Programmdatei ist mit einem Java Interpreter ausführbar und hat die Dateiendung “.class“.
Java Applets werden in HTML-Code eingebettet, d.h. es wird ein Verweis auf die Programmdatei des Applets gesetzt. Vor der Ausführung des Applets muß die Programmdatei von einem Server angefordert und über das Netzwerk übertragen werden (nur wenn Client und Server nicht auf einem Rechner installiert sind). Das HTML-Fragment, welches auf ein Applet verweist, kann folgendermaßen aussehen:

```
<APPLET code="Appletname.class" width=100 height=100>
<\APPLET>
```

Um das Applet auszuführen, muß auf dem Client eine *Java Virtual Machine* (JVM) installiert und aktiviert sein. Bei den gängigen Browsern (Netscape, Internet Explorer) ist dies der Fall.

## 2.2. Java Database Connectivity (JDBC)

JDBC wird benutzt, um auf relationale Datenbanken zuzugreifen und ist aus heutigen Anwendungen kaum noch wegzudenken, weil keine größere Anwendung ohne Datenbankanbindung auskommt. JDBC erlaubt einem Java-Programm SQL-Befehle (Structured Query Language) in einer Datenbank mit JDBC-Schnittstelle auszuführen. Es besteht die Möglichkeit, Anfragen (SELECT FROM WHERE) zu stellen und Antworten zu empfangen. Außerdem gibt es die Möglichkeit, Änderungen in einer Datenbank vorzunehmen (UPDATE, DELETE, INSERT, ....). In der JDBC-Spezifikation sind vier verschiedene Treibertypen festgeschrieben [1].

**Typ**1: **JDBC-ODBC-Bridge.** Dieser Typ nutzt einen auf dem Client installierten ODBC[1]-Treiber der spezifischen Datenbank. Diese Methode ist für das Internet eher ungeeignet, weil die Installation eines Treiber auf dem Client nötig ist.

**Typ**2: **Native API-Treiber.** Bei diesem Typ wird Java verwendet, um für die Verbindung zur Datenbank Funktionen eines Zugriffs-API aufzurufen. Auch dieser Typ ist auf einen installierten Treiber auf Client-Seite angewiesen.

**Typ**3: **Native Protokoll-Treiber.** Dieser Typ benutzt das Netzwerkprotokoll des Datenbankmanagementsystems, um eine direkte Verbindung zur Datenbank herzustellen. Diese Methode ist für Intranets geeignet, weil der Java-Treiber über das Netz nachgeladen werden kann und somit keine zusätzlichen Treiber auf dem Client vorhanden sein müssen. Ist der Client allerdings ein Applet, so wird verlangt, daß die Datenbank auf demselben Rechner zu erreichen ist, wie der Web-Server, von dem das Applet stammt. Bei dem Einsatz im Internet muß die Größe des Treibers und die zur Verfügung stehende Bandbreite berücksichtigt werden.

**Typ**4: **JDBC-Netztreiber.** Diese Variante benutzt die Netzwerkprotokolle des *Java Development Kit* (JDK), um eine Verbindung zu der Serverkomponente (der Teil der Anwendung, der auf dem Server liegt) aufzubauen. Auf dem Client müssen keine Treiber vorinstalliert sein und die Datenbank muß nicht auf dem selben Server liegen, wie die Serverkomponente und der Web-Server. Dieser Treibertyp ist sowohl für das Intranet als auch für das Internet geeignet.

[1]Open Database Connectivity

In dem folgenden Beispiel wird gezeigt, wie eine Verbindung zu einer Datenbank über eine JDBC-ODBC-Bridge herstellt werden kann.

```
import java.sql.*;
public class Datenbankzugriff {
public static void main(String args[]) {
 try {
  //Treibertyp festlegen
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 }catch (ClassNotFoundException e) {
   //Fehlermeldung ausgeben
   System.out.println("Kein Treiber gefunden!");
   return;
  }
 try{
  //Verbindung Connection-DriverManager
  Connection con = DriverManager.getConnetion
   ("jdbc:odbc:DatenbankName","","");
  //Verbindung Statement-Connection
  Statement stmt = con.createStatement();
  //Verbindung ResultSet-Statement
  ResultSet rs = stmt.executeQuery
   ("SELECT Vorname FROM Personendaten");
  //Das ResultSet wird durchlaufen
  while(rs.next) {
   System.out.println(rs.getString("Vorname"));
  }
  //Verbindungen schliessen
  rs.close();stmt.close();con.close();
  }
 } catch (SQLException se) {
    //Fehlermeldung ausgeben
    System.out.println(se.getMessage());
   }}
```

Die Funktionsweise des Programms läßt sich am besten mit Abbildung 1 erklären.

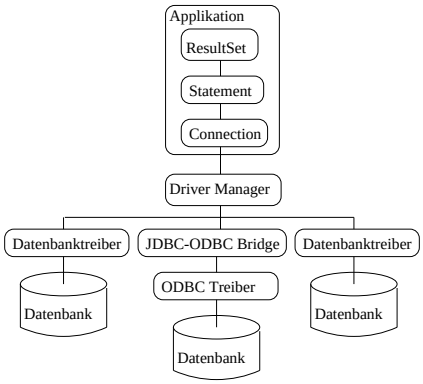


**Abbildung 1. JDBC Architektur**

Dort ist zu sehen, daß `Connection, Statement` und `ResultSet` voneinander abhängen. In der Variable `ResultSet` wird das Ergebnis der Anfrage gespeichert. In `Statement` wird die Anfrage festgelegt. Die `Connection` stellt die Verbindung zu `DriverManager`

her. Der `DriverManager` wiederum stellt die Verbindung zur Datenbank über den JDBC-Treiber (In Abbildung 1 als Datenbanktreiber bezeichnet.) oder die JDBC-ODBC-Bridge – wie im Beispiel – her. Großer Vorteil eines JDBC-Treibers ist, daß auf Datenbanken unterschiedlicher Hersteller zugegriffen werden kann, ohne für jede Datenbank individuell eine Schnittstelle programmieren zu müssen.
Weitere Information zu JDBC sind in [11] zu finden.

## 2.3. RMI (Remote Method Invocation)

RMI ermöglicht es Objekte, die auf einem Server ,,liegen", von einem Client aus zu benutzen. D.h., es können Funktionen von einem Client aus gestartet werden, so daß sie auf dem Server ausgeführt werden und das Ergebnis zurück zum Client geschickt wird. RMI kann nur innerhalb von Java benutzt werden, d.h. die Objekte auf dem Server sind Java-Objekte. Andere Systeme – wie CORBA – sind sprachunabhängig. Dort sind Verbindungen zwischen C, C++, Java, Smalltalk und Ada möglich.
Der Vorteil von RMI ist, daß die Objekte auf dem Server so behandelt werden können, als ob sie auf dem Client liegen würden. Die Kommunikation zwischen Client und Server wird von Java übernommen, d.h. sie ist für den Benutzer (Programmierer oder Anwendungsentwickler) transparent.

### RMI Architektur

Die RMI-Architektur setzt sich aus drei Schichten zusammen.

- Die *Stub/Skeleton-Schicht*, welche eine Schnittstelle für die Client- und Server-Objekte zur Verfügung stellt, um miteinander zu kommunizieren.
- Die *Remote Reference-Schicht*, welche als Middleware zwischen Stub/Skeleton-Schicht und dem Transportprotokoll liegt. Diese Schicht regelt den Fernzugriff auf die Objektreferenzen.
- Die *Transportprotokoll-Schicht*, welche den eigentlichen Transport über das Netzwerk regelt.

Eine schematische Darstellung ist in Abbildung 2 zu sehen.

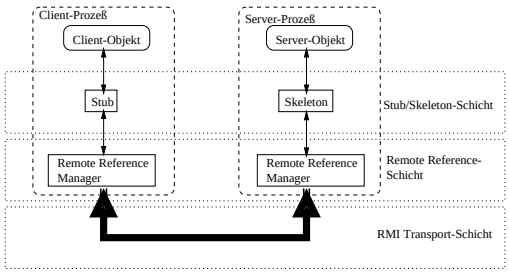


**Abbildung 2. RMI Architektur**

In der Abbildung ist zu sehen, daß der Client eine Anfrage auf ein Server-Objekt mit Hilfe des Stub startet: Dies geschied durch Starten einer Anfrage mittels Aufruf einer Stub-Objekt-Methode. Der Stub ,,packt" die Argumente der Funktion zusammen und leitet sie an den Remote Reference Manager des Client weiter. Von dort wird sie über das Netzwerk zum Remote Reference Manager der Serverseite übertragen. Der Remote Reference Manager des Servers leitet die Argumente an den Server-Skeleton weiter. Im Skeleton werden die Argumente wieder ,,entpackt" und an das Server-Objekt weitergeleitet und dort ausgeführt. Der Rückgabewert gelangt auf dem gleichen Weg zurück zum Client.
Die Implementierung und die Erzeugung von Stub und Skeleton sowie die Kommunikation zwischen den Objekten lassen sich am besten an einem Beispiel verdeutlichen, das auch die Abläufe im einzelnen erklärt.
Es ist zum Beispiel vorstellbar, daß jemand ein ,,schwarzes" Konto in Liechtenstein oder der Schweiz eröffnet, auf das über einen Client (Bankschalter) zugegriffen werden kann. Das Konto liegt dabei auf dem Server (Bankrechner).
Um RMI zu nutzen, werden folgende Klassen und Interfaces benötigt:

1. ein Interface mit Funktionen
2. eine Implementierung des Remote Interface
3. eine Klasse, die ein Objekt erzeugt, das auf dem Server liegt und von dem Client benutzt werden kann
4. eine Client-Klasse, die die vom Server bereitgestellten Funktionen benutzt

1. Das Interface definiert die benötigten Funktionen, die dem Client zur Verfügung gestellt werden sollen. Im vorliegenden Beispiel werden nur die Funktionen `Kontostand` und `Einzahlung` in das Interface aufgenommen und implementiert. In der Realität sind Funktionen wie: ,,Überweisung", ,,Abheben", ,,Kontoinhaber", usw. auch denkbar.

```
import java.rmi.Remote;
import java.rmi.RemoteException;
 public interface Account extends Remote {
 public void Einzahlung(float Betrag)
   throws RemoteException;
public float Kontostand()throws RemoteException;
}
```

Jede Funktion erzeugt eine Exception, falls ein Fehler auftritt.
2. Das unter 1. festgelegte Interface muß implementiert werden.

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
public class AccountImpl
 extends UnicastRemoteObject implements Account {
 //Anfangskontostand
 private float Konto = 0;
 public void Einzahlung(float Betrag)
     throws RemoteException {
  //Kontostand erhoehen
  Konto = Konto + Betrag;
 }
```

```
 public float Kontostand()
   throws RemoteException {
  //Kontostand zurueckgeben
  return Konto;
}}
```

Mit dem Befehl `rmic AccountImpl` werden die Stub- und Skeleton-Klasse erzeugt.
3. Um ein Objekt auf dem Server zu erzeugen, wird eine eigene Klasse erstellt. Diese Klasse wird auf dem Server ausgeführt, um das Konto-Objekt zu erzeugen.

```
import java.rmi.Naming;
public class RegAccount {
 public static void main(String args[]) {
  try {
   //Konto anlegen
   AccountImpl Konto = new AccountImpl("HelmutK");
   //Konto anmelden
   Naming.rebind("HelmutK", Konto);
  }
  catch (Exception e) {
   //Fehlermeldung ausgeben
   e.printStackTrace();
}}}
```

In der Klasse `RegAccount` wird ein neues Konto eröffnet und bei der RMI Registry angemeldet. Dazu wird das Interface von `java.rmi.Naming` benutzt. Die Anmeldung ist nötig, um das Server-Objekt im Netzwerk zu finden.
Bevor das kompilierte Programm `RegAccount` mit `java RegAccount` gestartet wird, muß die RMI Registry gestartet werden. Dies geschieht unter UNIX mit dem Befehl: `rmiregistry &`. Unter Windows, in einem DOS-Fenster, mit: `start rmiregistry`.
4. Als letztes muß ein Client erstellt werden.

```
import java.rmi.Naming;
public class AccountClient {
 public static void main(String args[]) {
  try{
   //Account anlegen
   Account GSAccount;
   //suchen des Server-Objekts ''HelmutK''
   //auf Server ''Server1''
   GSAccount =
     (Account)Naming.lookup("rmi://Server1/HelmutK");
   //12000 DM einzahlen
   GSAccount.Einzahlung(12000);
   //Kontostand ausgeben
   System.out.println("Ihr Kontostand ist:"+
    GSAccount.Kontostand());
  }
  catch (Exception e) {
   //Fehlermeldung ausgeben
   e.printStackTrace();
}}}
```

Diese Klasse sucht das angegebene Konto in der RMI Registry. Falls es eingetragen ist, wird eine Einzahlung vorgenommen.
Das Zusammenspiel der einzelnen Komponenten wird in Abbildung 3 verdeutlicht. Ein weiteres Beispiel findet sich in [9]. Weiterführende Informationen sind in [1] und [10] zu finden.

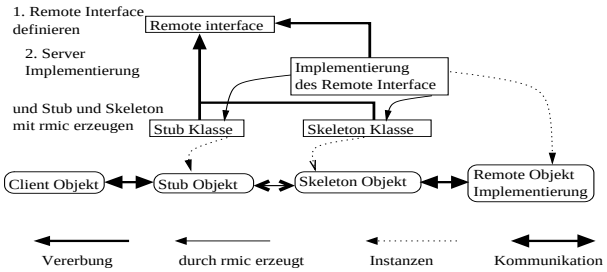


**Abbildung 3. Zusammenspiel der Objekte und Klassen**

## 2.4. Java IDL und CORBA

In diesem Abschnitt wird ein kurzer Überblick über Java IDL und CORBA gegeben.

### 2.4.1. Java IDL

Die Java IDL API bietet ab der Version Java 1.2 ein Interface zwischen Java und verteilten Objekten oder Services, die auf CORBA basieren, an. CORBA ist ein Standard, der von der Object Management Group (OMG) [5] definiert wurde. Der Standard beschreibt eine Architektur von Schnittstellen und Protokollen, die verteilte Objekte benutzen können, um miteinander zu kommunizieren. Ein Teil des CORBA-Standards ist die Interface Definition Language (IDL), die es ermöglicht, Interfaces für entfernte Objekte zu beschreiben. Es wurden Standardumsetzungen von IDL in C++ Klassen, C-Code und Java-Klassen festgelegt. Diese generierten Klassen benutzen CORBA und bilden die Grundlage der Kommunikation zwischen verteilten Objekten.
Wie RMI, bietet auch IDL die Möglichkeit, entfernte Objekte über ein Netzwerk anzusprechen. So kann z.B. ein Client eine Methode eines entfernten Objekts ausführen und auch Daten von dem Objekt empfangen. Für den Benutzer ist dieser Vorgang transparent, d.h. er greift auf ein Objekt zu, als ob es lokal auf seinem Rechner, vorhanden wäre. Der Vorteil gegenüber RMI ist, daß der Client auch in anderen Sprachen programmiert werden kann, z.B. C++, C oder Ada.

### 2.4.2. CORBA

Der CORBA-Standard ist sehr umfangreich. Neben der Basisarchitektur und der Syntax von IDL, sind auch verschiedene Dienste, die von Objekten angeboten werden, wie der Anmeldeservice und Sicherheitsbeschränkungen, in dem Standard festgeschrieben. Dieser Artikel behandelt

nur die Basisarchitektur.
Die CORBA-Architektur hat viele Ähnlichkeiten mit der von RMI. Die Beschreibung (Interface) eines entfernten Objekts wird benutzt, um einen Client-Stub und ein Server-Skeleton zu erzeugen. Der Client benutzt den Stub als Interface, um Methoden auf dem entfernten Objekt aufzurufen. Der Funktionsaufruf wird mit Attributen über das Netzwerk zum Server-Skeleton übertragen. Dieser leitet den Aufruf weiter an das Server-Objekt. Ergebnisse oder Fehler werden auf dem gleichen Weg, nur in umgekehrter Richtung, übertragen.

**Objekt Request Broker**
Der *Object Request Broker* (ORB) ist ein sehr wichtiger Teil der CORBA-Architektur. In der Abbildung 4 ist eine schematische Darstellung zu sehen.

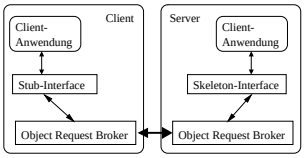


**Abbildung 4. CORBA Architektur**

Jede Anwendung, die CORBA benutzt, benötigt einen ORB damit die verteilten Objekte miteinander kommunizieren können. Anfragen vom Client werden vom Stub an den Client ORB geleitet, der sie über das Netzwerk an den Server ORB weiterleitet. Auf dem Server werden sie vom ORB über das Skeleton an das Server-Objekt geschickt, das die Anfrage bearbeitet. Ebenso ist der ORB auch für die Weiterleitung der Fehlermeldungen an den Client-Stub zuständig. Eine genauere Einführung in CORBA findet man in [7] und [8].

## 2.5. Java Servlets

Java Servlets sind geeignet, um die Funktionalität eines Servers zu erweitern. Sie werden normalerweise auf Web-Servern ausgeführt, wo sie in der Lage sind, Common Gateway Interface Skripts (CGI-Skripts) zu ersetzen. Dies bedeutet, daß man mit Servlets dynamische Webinhalte erzeugen kann. Servlets sind rechnerunabhängig und schneller als CGI-Skripts [1]. Außerdem stehen Servlets alle APIs von Java zur Verfügung, eingeschlossen JDBC, womit ein problemloser Datenbankzugriff ermöglicht wird. Servlets haben im Allgemeinen keine Sicherheitsbeschränkungen, d.h. ein Servlet hat z.B. Zugriff auf das Dateisystem des Servers. Die Sicherheitsbeschränkungen werden typischerweise durch den Web-Server geregelt.

### 2.5.1. Architektur von Web-Servern mit Servlets

Ein Client ruft eine HTML-Seite auf, in die ein Verweis auf ein Servlet integriert ist. Die Anfrage geht beim Web-Server ein, der die entsprechenden Java-Klassen aufruft und ein Ergebnis berechnet. Dieses wird an den Client zurücksendet. Meistens ist das Ergebnis einer solchen Anfrage wieder eine HTML-Seite. Für den Client ist dieser Vorgang völlig transparent, d.h. er bemerkt nicht, daß ein Servlet die HTML-Seite erstellt hat.
Der oben angesprochene Geschwindigkeitsvorteil ergibt sich daraus, daß die Servlets nicht notwendigerweise beendet werden, nachdem sie das Ergebnis geliefert haben, d.h. sie können weiterhin aktiv sein und auf Anfragen warten. Damit entfällt das erneute Initialisieren und Starten des Servlets. Verdeutlicht wird dieser Ablauf in Abbildung 5.

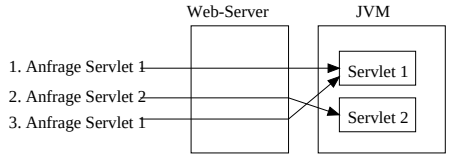


**Abbildung 5. Lebensdauer eines Servlets**

Es ist zu sehen, daß Anfrage 1 Servlet 1 aufruft. Servlet 1 bleibt während Anfrage 2 aktiv. Bei Anfrage 3 muß Servlet 1 nicht gestartet werden, sondern es kann das noch aktive Servlet 1 benutzt werden.

Im folgenden ist ein HTML-Fragment zum Einbinden eines Servlets zu sehen.

```
<FORM METHOD=GET ACTION="NameServlet">
Bitte Vornamen eingeben:
<INPUT TYPE=TEXT NAME=Vorname SIZE=30>
<INPUT TYPE=SUBMIT VALUE="Abschicken">
</FORM>
```

In ihm wird ein Eingabefeld definiert, das einen Wert `Vorname` einliest, der durch Betätigen des „Abschicken-Button" an das Servlet übermittelt wird.
Die folgenden Zeilen zeigen den Quellcode des Servlets.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class NameServlet extends HttpServlet {
 public void doGet(HttpServletRequest req,
  HttpServletResponse resp)
  throws ServletException, IOException {
  resp.setContentType("text/html");
  PrintWriter out = resp.getWriter();
  out.println("<HTML>");
  out.println("<HEAD><TITLE>Gruss</TITLE></HEAD>");
  out.println("<BODY>Hallo, "+
    req.getParameter("Vorname"));
  out.println("</BODY></HTML>");
 }}
```

Dieses kleine Beispielprogramm erzeugt ein Stück HTML-Code und gibt es an den Client zurück. Mit `setContentType` wird der Ausgabetyp festgelegt. `getParameter` liest aus dem `HttpServletRequest` den Parameter `Vorname` aus und fügt ihn in den HTML-Code ein.
Weitere Informationen und Beispiele sind in [4] zu finden.

## 2.6. Java Naming and Directory Interface (JNDI)

JNDI soll der Vollständigkeit wegen erwähnt und nur kurz erklärt werden.
Er ermöglicht das Anmelden und Suchen von Objekten in einem Netzwerk. JNDI kann z.B. auch in Verbindungen mit RMI, wo Objekte durch RMI-Registry angemeldet werden, zum Einsatz kommen. Es wird auch dazu benutzt, um Namensdienste, wie DNS (Domain Name Server), LDAP (Lightweight Directory Access Protocoll) oder NDS (Novell Directory Service), anzusprechen.
Wie aus dem Namen JNDI zu entnehmen ist, enthält JNDI nicht nur einen Anmeldedienst (Naming), sondern auch einen Verzeichnisdienst (Directory). Der Unterschied zum Anmeldedienst ist, daß es bei Verzeichnissen auch möglich ist, Attribute mit Objekten zu verbinden. Auf diese Weise ist es möglich, Objekte nicht nur nach Namen zu suchen, sondern auch nach Attributen.

## 2.7. JavaBeans (JB) und Enterprise JavaBeans (EJB)

Um EJB zu verstehen, muß erst die Funktionsweise von JB erklärt werden.

### 2.7.1. JavaBeans (JB)

JB sind ein Softwarekomponentenmodell für Java. Mit JB können wiederverwendbare Softwarekomponenten entwickelt werden können, die mit einem Generierungstool, wie Bean Development Kit (BDK), visuell manipuliert werden können. Sie können allerdings auch ohne Generierungstool in ein Programm integriert werden. Häufig werden JB beim Programmieren von Benutzeroberflächen eingesetzt. Sie können in Applets, Applikationen und Servlets integriert werden.
JB können von drei unterschiedlichen Programmierertypen benutzt werden:

- Ebene 1: Wenn GUI-Editoren, Anwendungsgeneratoren oder ,,Beanbox-Tools" [2] entwickelt werden sollen, dann wird die JB-API benötigt, um die Beans zu manipulieren.
- Ebene 2: Wenn man JB entwickelt, dann benötigt man die JB-API, um Programme zu schreiben, die in einer ,,beanbox" benutzt werden können.
- Ebene 3: Wenn man Programme schreibt, die JB, die von anderen Programmierern erstellt wurden, benutzt, dann muß man sich nicht mit der JB-API auseinandersetzen. Man muß nur die vom JB zur Verfügung gestellten Methoden kennen.

Um mit JB der Ebene 1 arbeiten zu können, wird z.B. das Beanbox-Tool des BDK benötigt. Mit diesem Tool kann bespielsweise ein Java-Button erstellt werden. Allerdings wird der Java-Button nicht ,,per Hand" programmiert, sondern durch ,,Drag and Drop". Das Beanbox-Tool stellt normalerweise drei Fenster zur Verfügung: Toolbox-, Properties- und Beanbox-Fenster. In dem Toolbox-Fenster sind die verfügbaren JB gelistet, die mit ,,Drag and Drop" im Beanbox-Fenster plaziert werden können. Im Properties-Fenster können die Eigenschaften der JB verändert werden. Auf einen Button bezogen, bedeutet dies, daß z.B. die Schriftart und -größe verändert werden kann. In Abbildung 6 ist dies zu sehen.

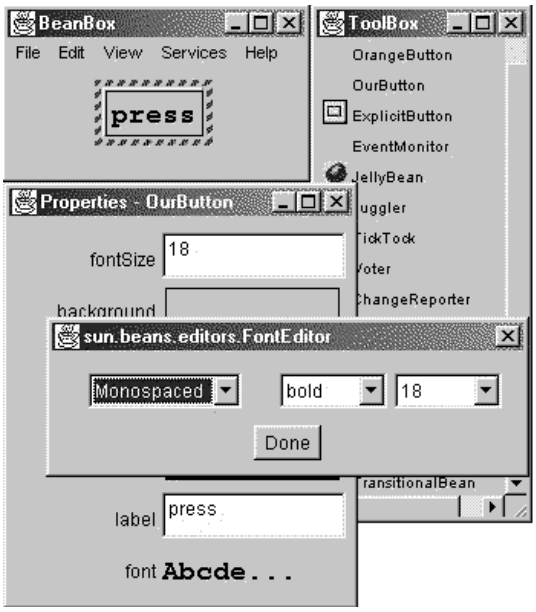


**Abbildung 6. JavaBeans**

Im folgenden wird ein kurzes Beispielprogramm erläutert, um den generellen Aubau eines JB zu erklären. Dies wäre die Entwicklung von JB auch Ebene 2.

```
public class SimpleBean extends Canvas
                    implements Serializable{
  private Color color = Color.green;
  //get-Methode
  public Color getColor(){ return color;}
  //set-Methode
```

[2] Unter einem ,,Beanbox-Tool" versteht man ein einfaches JB-Tool, welches im BDK enthalten ist.

```
public void setColor(Color newColor){
   color = newColor; repaint();
}
public void paint(Graphics g) {
 g.setColor(color); g.fillRect(20, 5, 20, 30);
}
//Konstruktor setzt die geerbten Eigenschaften
public SimpleBean(){
 setSize(60,40); setBackground(Color.red);
}}
```

Wie in dem Beispiel zu sehen ist, werden in einem JB set- und get-Methoden zur Verfügung gestellt, um Eigenschaft des Objekts zu ändern. Diese Methoden sind die einzige Möglichkeit Eigenschaften eines Objekts zu ändern. Mit den zur Verfügung gestellten Methoden kann die Farbe eines Rechtecks (Rectangle) gesetzt und ausgelesen werden. Dieses JB kann der Beanbox zur Verfügung gestellt werden, um bei einem Rechteck die Farbe festzulegen.
Wenn JB auf Ebene 3 verwendet werden, kann muß nur die Schnittstelle bekannt sein. Auf das oben angegebene Beispiel bezogen, mußsen den Programmierer nur die set- und get-Methoden bekannt sein, um das JB zu benutzen.

#### 2.7.2. Enterprise JavaBeans (EJB)

EJB bauen auf JB und RMI auf und bietet einen Standard für verteilte Objekt-Komponenten. EJB bestehen allerdings allerdings nicht aus JB, die mittels RMI über ein Netzwerk angesprochen werden, sondern die Architektur wurde auch um Transaktionsverarbeitung (Transaction Processing), Sicherheitskonzepte (Security), Zustandsspeicherung (Persistence) und Resourcen Teilung (Rescource Pooling) erweitert.
Damit ist es mit EJB möglich umfangreiche und komplexe Anwendungen zu erstellen, die über ein Netzwerk kommunizieren. Weiterführende Informationen sind in [2] zu finden.

## 3. Vergleich Applets und Servlets

In diesem Abschnitt sollen Applets und Servlets verglichen, weil sich in der Praxis oft die Frage stellt, welcher Ansatz für die Problemlösung geeigneter ist. Hauptaugenmerk beim Vergleich liegt in der Geschwindigkeit, dem Ort der Programmausführung und den Sicherheitsbeschränkungen.
Ein Applet wird auf einem Client ausgeführt. Es muß vor der Ausführung von dem Server angefordert und über das Netzwerk übertragen werden. Bei einem Servlet wird eine Anfrage an den Server gestellt und das Servlet wird auf dem Server ausgeführt. Nur das Ergebnis wird über das Netzwerk zurück zum Client übertragen. Veranschaulicht wird dies in Abbildung 7.

Daraus ergeben sich Geschwindigkeitsnachteile für das Applet, da es über das Netzwerk übertragen und die JVM

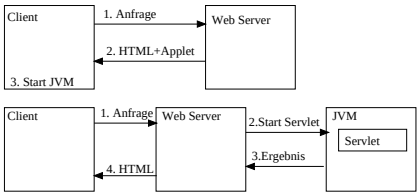


**Abbildung 7. Client Server**

im Browser gestartet werden muß.
Die Sicherheitsbeschränkungen sind bei Applets viel strenger als bei Servlets. Applets haben im Normalfall keinen Zugriff auf das Dateisystem und sind nicht berechtigt Netzwerkoperationen auszuführen. Es gibt noch eine Reihe Beschränkungen, die sich hauptsächlich auf Systemeigenschaften beziehen und hier nicht weiter erläutert werden. Bei Servlets werden die Beschränkungen durch den Web-Server bestimmt, d.h. dem Servlet ist alles erlaubt (Dateizugriff, ...), was der Web-Server nicht verbietet.
Bei dem Vergleich sollte aber nicht aus den Augen verloren werden, was mit einem Programm erreicht werden soll. Soll eine einfache Datenbankanfrage gestellt werden (z.B. Suchen eines Buches bei Amazon), dann ist sicher ein Servlet besser geeignet. Wenn aufwendige Datenbankoperationen von einem Client durchgeführt werden sollen (z.B. Pflege einer Datenbank), dann ist ein Applet oder eine Applikation sinnvoller. Deutlicher wird dieser Sachverhalt in Abschnitt 4.

## 4. Beispielanwendung mit Java

In diesem Abschnitt soll an einem Beispiel gezeigt werden, wie mit den oben beschriebenen Java-Komponenten eine Client/Server-Anwendung erstellt werden kann. In dem Beispiel wird eine Internetbuchhandlung beschrieben. Die Architektur einer solchen Anwendung kann, wie in Abbildung 8 dargestellt, aussehen.

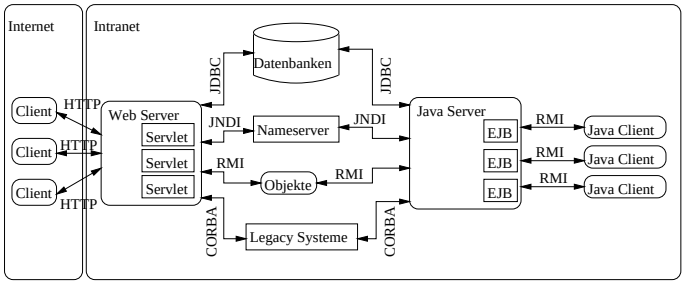


**Abbildung 8. Beispielanwendung**

a.)Aus der Sicht des Kunden

- Der Kunde greift mit seinem Webbrowser auf den Webserver zu, um Informationen über ein Buch zu erhalten oder ein Buch zu bestellen.
- Der Web-Server empfängt die Anfragen von dem Kunden und leitet sie an das entsprechende Servlet weiter.
- Anfragen bezüglich Büchern werden über die JDBC-Schnittstelle vom Servlet an eine Datenbank weitergeleitet, um die gewünschten Informationen bereitzustellen.
- Personendaten, die der Kunde bei einer Bestellungen übermittelt, werden mit Hilfe von CORBA-Objekten an ein Buchhaltungsprogramm weitergeleitet, welches in C programmiert ist.
- JNDI wird benutzt, um herauszufinden in welchem Lager das gewünschte Buch noch vorrätig ist.
- Wenn der Kunde Bücher in seinen virtuellen Einkaufswagen legen möchte, wird ein Objekt angelegt, welches mit RMI angesprochen wird.

b.)Aus der Sicht des Händlers

- Der Händler benutzt einen Java-Client, welcher als Applet oder als Applikation realisiert sein kann, um Daten zu verwalten.
- Der Java-Client kommuniziert mit einem Java-Server mittels RMI. Der Server ist mit EJB realisiert.
- Um den Buchbestand in der Datenbank zu erweitern, kommuniziert der Server über JDBC mit den verschiedenen Datenbanken.
- Um auf die Name Server zuzugreifen, verwendet der Server JNDI.
- Will der Verkäufer für statistische Zwecke überwachen, welche Bücher der Kunde in seinen virtuellen Einkaufswagen legt, so wird dies mittels RMI realisiert.
- Um Kundendaten zu pflegen, kommuniziert der Java-Server mittels CORBA-Objekten mit der Buchhaltungssoftware.

Dies ist natürlich ein etwas konstruiertes Beispiel, aber es verdeutlicht, wie die einzelnen Java Komponenten eingesetzt werden können, um Client/Server Anwendungen zu erstellen.

## 5. Fazit/Zusammenfassung/Ausblick

Diesem Artikel sollte verdeutlichen, daß Java ein mächtiges Werkzeug ist, um Client/Server-Anwendungen zu erstellen. Die Vorteile von Java sind, daß eine Reihe von Konzepten und Schnittstellen angeboten werden, und so nicht mehr selbst programmiert werden müssen. So ist es vergleichsweise einfach möglich, eine Client/Server-Kommunikation mit RMI zu realisieren. Mit anderen Programmiersprachen, wie z.B. C oder C++ müssen solche Schnittstellen selbst programmiert oder gekauft werden. Deshalb ist die Fehleranfälligkeit und die Entwicklungszeit bei Java erheblich kürzer und damit billiger. Außerdem ist es mit Java möglich, Programme, die in anderen Programmiersprachen erstellt wurden, mittels IDL und CORBA anzusprechen. Auch dadurch werden Ressourcen gespart.
Ein weiterer Pluspunkt von Java ist, daß plattformübergreifend entwickelt werden kann, d.h. man schreibt nur ein Programm, welches dann auf allen gängigen Betriebssystemen, wie Windows- , Unix- oder Linux, läuft.
Allerdings sollte bei allen positiven Aspekten von Java auch erwähnt werden, daß Java in der Programmausführung erheblich langsamer als beispielsweise C oder C++ ist.
Für die Zukunft kann man sagen, daß Java wahrscheinlich weiter an Bedeutung gewinnen wird, weil es – wie oben schon gesagt – verhältnismäßig einfach ist, Client/Server-Anwendungen zu erstellen [1]. Ein weiteres Indiz dafür ist, daß Sun RMI auf reiner Java-Basis in die Standardplattform (JDK 1.3) integriert hat. Auch JNDI gehört jetzt zur Standardplattform[6].

## Literatur


[1] Flanagan D., Farley J., Crawford W., Magnusson K., ”Java Enterprise in a Nutshell”, *O’Reilly*, 1999, ISBN 1565924835

[2] Denninger S., ”Enterprise JavaBeans”, *Addision-Wesley*, 2000, ISBN 3827315344

[3] Flanagan D., ”Java in a Nutshell”, *O’Reilly*, 1998, ISBN 3897211009

[4] Hunter J., ”Java Servlet Programming”, *O’Reilly*, 1998, ISBN 156592391X

[5] Object Management Group: www.omg.org

[6] Menge R., ”c’t 11/00 Sanfte Java-Entwicklung; Überholt: JDK 1.3”, *Heise-Verlag*, 2000

[7] Sayegh, A., ”Corba. Standard, Spezifikation, Entwicklung”. *O’Reilly*, 1999, ISBN 3897211564

[8] Harkey D., Orfali R., ”Client/Server Programming with Java and CORBA, Second Edition” *Wiley*, 1998, ISBN 047124578X

[9] Boger M., ”Java in verteilten Systemen” *dpunkt-Verlag*, 1999, ISBN 3932588320

[10] Schulz K., ”Java professionell programmieren” *Springer-Verlag*, 1999, ISBN 354065710X

[11] Dehnhardt W., ”Anwendungsprogrammierung mit JDBC” *Carl Hanser*, 1999, ISBN 3446212655

[12] Patzer A.,”Professional Java Server Programming: with Servlets, JSP, XML, EJB, JNDI, CORBA, Jini and JavaSpaces” *Wrox Press*, 1999, ISBN 1861002777