Dietmar Abts

Masterkurs Client/Server-Programmierung mit Java

Leserstimmen zur 1. Auflage:

„Sehr schöne Beispiele. Sehr aktuelle Themen."
Prof. Dr. Reinhard Brocks, HTW Saarbrücken

„Besonders gut gefallen mir die vielen Programmbeispiele und guten Übungsaufgaben."
Prof. Dr. Helmut Jarosch, FHW Berlin

„Beispielhafte, gut verständliche Darstellung einiger nicht trivialen Möglichkeiten von Java..."
Prof. Dr. G. Klein, FH Furtwangen

„Ausführliche Beispiele, die für vertiefende Experimente als Quellcode über den Online-Service zum Buch verfügbar sind."
Prof. Dr. Bernd Steinbach, TU BAF, Freiberg

„Praxisnahe Darstellung mit anschaulichen Beispielen und Übungen."
ez-Informationsdienst, 24/03

Dietmar Abts

Masterkurs Client/Server-Programmierung mit Java

Anwendungen entwickeln mit JDBC, Sockets, XML-RPC, RMI und JMS – Kompakt und praxisnah – Zahlreiche Programmbeispiele und Aufgaben

2., erweiterte und aktualisierte Auflage

Mit 68 Abbildungen

vieweg

Bibliografische Information Der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

1. Auflage 2003
Diese Auflage erschien unter dem Titel „Aufbaukurs JAVA“
2., erweiterte und aktualisierte Auflage April 2007

Lektorat: Sybille Thelen / Andrea Broßler

Der Vieweg Verlag ist ein Unternehmen von Springer Science+Business Media.
www.vieweg.de

Umschlaggestaltung: Ulrike Weigel, www.CorporateDesignGroup.de
Umschlagbild: Nina Faber de.sign, Wiesbaden
Druck- und buchbinderische Verarbeitung: MercedesDruck, Berlin
Printed in Germany

ISBN 978-3-8348-0322-1

Vorwort zur zweiten Auflage

Lange Zeit wurde die betriebliche Datenverarbeitung durch monolithische Anwendungsprogramme auf Großrechnern bestimmt. Mit dem Aufkommen von preiswerten mittleren Rechnersystemen und PCs begann ein starker Trend zur Dezentralisierung. Heute sind verteilte Anwendungen üblich, deren Komponenten auf unterschiedlichen Rechnern im Netz miteinander kooperieren und dabei die Ressourcen der beteiligten Systeme optimal ausnutzen. Eine sehr verbreitete Anwendung dieser Art ist das World Wide Web. Auf der anderen Seite stellen aber verteilte Systeme hohe Anforderungen an die Softwareentwickler. Diese müssen sich mit Problemen der Kommunikation und Koordination nebenläufiger Prozesse in einem heterogenen Umfeld beschäftigen. Moderne Programmierkonzepte und eine ausgereifte technische Infrastruktur (Middleware) bieten Unterstützung bei der Entwicklung verteilter Anwendungen.

Der vorliegende Masterkurs bietet eine kompakte Einführung in aktuelle und zukunftsweisende Technologien auf der Basis von Java SE ab Version 5.0 mit zahlreichen Programmbeispielen und Übungsaufgaben. Die verschiedenen Themen können mit Basiskenntnissen der Programmiersprache Java erarbeitet werden.

Gegenüber der ersten Auflage mit dem Titel "Aufbaukurs Java" ist ein Kapitel über nachrichtenorientierte Middleware auf der Basis von JMS hinzugekommen. Um die Thematik des Buches im Titel besser auszudrücken, wurde dieser für die neue Auflage geändert: "Masterkurs Client/Server-Programmierung mit Java". Alle Beispiele wurden komplett überarbeitet und teilweise durch neue ersetzt. Sie sind für die aktuellen Versionen der hier eingesetzten Produkte (MySQL, Apache Derby, Apache XML-RPC, Apache Tomcat, JBoss AS, Apache ActiveMQ) lauffähig.

Der Quellcode aller Programme sowie die Lösungen zu den Übungsaufgaben liegen im Internet zum Download bereit. Den Zugang zu diesen Zusatzmaterialien finden Sie auf der Website des Verlags

```
www.vieweg.de
```

direkt neben den bibliographischen Angaben zu diesem Buch.

Viele Erfahrungen, die ich im Rahmen einer vier Semesterwochenstunden umfassenden Lehrveranstaltung zum Thema "Verteilte Systeme" gewonnen habe, sind in dieses Buch eingeflossen.

Ich hoffe, dass Sie viel Spaß bei der Erarbeitung des Stoffes und der Entwicklung eigener Programme haben. Anregungen und Hinweise senden Sie bitte an die folgende Adresse:

abts@hs-niederrhein.de

Ratingen, März 2007 Dietmar Abts

Inhaltsverzeichnis

1 Einleitung und Grundlagen

1.1 Vorbemerkungen

Java wird heute als sehr gut geeignete Programmiersprache für große Internet- und Intranet-Anwendungen insbesondere auf der Serverseite eingesetzt. Die Plattformunabhängigkeit, reichhaltige Klassenbibliotheken und die Unterstützung zahlreicher APIs (Application Programming Interfaces) macht Java zur bevorzugten Sprache für verteilte Anwendungen in einem heterogenen Umfeld.

Zielsetzung

Das vorliegende Buch beschäftigt sich mit der Implementierung von *Client/Server-Anwendungen* auf Basis der Internet-Protokolle. Angesichts des Umfangs der Themen musste eine Auswahl getroffen werden, die auch in einem vier Semesterwochenstunden umfassenden Kurs behandelt werden kann.

Ziel des Buches ist es, über die Themenvielfalt angemessen zu informieren und in die Einzelthemen systematisch mit der nötigen Tiefe einzuführen. Besonderer Wert wurde dabei auf praxisnahe Programmbeispiele und Übungsaufgaben gelegt.

Aufbau des Buches

Dieses Buch gliedert sich in acht Kapitel, die jeweils einen Schwerpunkt behandeln. Obwohl die Kapitel weitestgehend in sich geschlossen sind und unabhängig voneinander bearbeitet werden können, wird empfohlen, diese in der hier vorgegebenen Reihenfolge durchzuarbeiten. Die vorgestellten Themen werden anhand vollständiger, lauffähiger Programmbeispiele verdeutlicht. Übungsaufgaben am Ende eines jeden Kapitels sollen zur selbständigen Beschäftigung mit dem dargebotenen Stoff anregen.

Das Buch hat folgende Kapitel:

1 Einleitung und Grundlagen
Dieses Kapitel gibt eine theoretische Einführung in die *verteilte Verarbeitung* auf der Basis des *Client/Server-Modells* und stellt *mehrstufige Architekturen* vor. Sinn und Zweck von *Middleware* werden an einem ersten Programmbeispiel erläutert. Hier werden auch grundlegende Begriffe des Internets (wie IP-Adressen, Adressauflösung), die zum Verständnis der übrigen Kapitel nötig sind, erklärt.

2 Datenbankanwendungen mit JDBC
In mehrstufigen Architekturen verteilter Systeme, so z.B. bei *dynamischen Webanwendungen*, erzeugt der Applikationsserver Datenbankabfragen und schickt sie zur Ausführung an den Datenbankserver. *JDBC* ist das Standard-API für den Zugriff auf relationale Datenbanken mittels *SQL-Anwei-*

sungen, die zur Laufzeit dynamisch aufgebaut werden können. Wir beschäftigen uns mit den Grundlagen von JDBC, die interessante Client/Server-Anwendungen in den folgenden Kapiteln ermöglichen. Ein Exkurs zeigt, wie das Ergebnis einer SQL-Abfrage in ein XML- bzw. HTML-Dokument transformiert werden kann. Ein weiterer Exkurs gibt eine kurze Einführung in die JDBC-Schnittstelle zu *Stored Procedures* am Beispiel von MySQL.

3 Verbindungslose Kommunikation mit UDP

Das *User Datagram Protocol* (UDP) ist neben TCP ein wichtiges Transportprotokoll im Internet. Es ermöglicht, mit geringem Aufwand Datenpakete zu versenden, ohne vorher eine Verbindung zum Empfänger aufbauen zu müssen. Wir entwickeln u.a. ein Programm, mit dem zwei Benutzer an verschiedenen Rechnern im Netz eine Unterhaltung online führen können.

4 Client/Server-Anwendungen mit TCP

Fast alle Internet-Dienste nutzen das *Transmission Control Protocol* (TCP). Im Gegensatz zu UDP stellt TCP eine verlässliche Ende-zu-Ende-Verbindung zwischen Sender und Empfänger her. Der Anwendungsentwickler muss sich bei der Übertragung eines Datenstroms nicht um die Aufteilung in einzelne Pakete kümmern. *Sockets* sind die Endpunkte einer TCP-Verbindung. Sie ermöglichen es, das Senden und Empfangen von Daten wie das Schreiben und Lesen von Dateien zu behandeln. *Multithreading* versetzt den Server in die Lage, mehrere Client-Anfragen gleichzeitig zu bearbeiten. Wir entwickeln u.a. ein Dateiübertragungsprogramm sowie eine Anwendung, mit der ein so genanntes "Chatten" innerhalb einer Gruppe von Teilnehmern möglich ist. Der Einsatz wiederverwendbarer Threads eines *Thread-Pools* reduziert den Ressourcenverbrauch und kann die Effizienz der Anwendung erhöhen. Das Kapitel wird mit der Vorstellung eines *Frameworks* für TCP-Server abgeschlossen.

5 Implementierung eines HTTP-Servers

Dieses Kapitel behandelt die Grundlagen des Anwendungsprotokolls *Hypertext Transfer Protocol* (HTTP), das die Kommunikationsfunktionalität des *World Wide Web* definiert. HTTP verwendet auf der Transportschicht TCP. Wir implementieren u.a. einen speziellen HTTP-Server, der beliebige SQL-Anweisungen vom Webbrowser entgegennimmt und ausführt, sowie einen einfachen Webserver zur Übertragung von statischen Webseiten und anderen Ressourcen. Dieser Webserver wird dann so erweitert, dass anwendungsspezifische Klassen zur Laufzeit nachgeladen werden können, um mit deren Hilfe Webseiten ad hoc zu generieren.

6 XML Remote Procedure Call (XML-RPC)
In diesem Kapitel behandeln wir *Remote Procedure Calls* (RPC) und zeigen anhand des *XML-RPC*-Verfahrens, wie HTTP und XML als Datenaustauschformat für den entfernten Prozeduraufruf eingesetzt werden können. Auf dieser Basis implementieren wir einfache *Web Services* und zeigen, wie die XML-RPC-Verarbeitung in *Apache Tomcat* eingebunden werden kann.

7 Entfernter Methodenaufruf mit RMI
Remote Method Invocation (RMI) ist eine einfache und elegante Art, Client/Server-Anwendungen in Java zu implementieren. Der Entwickler kann sich ganz auf die fachliche Funktionalität konzentrieren, netzspezifische Details bleiben ihm im Gegensatz zur Socket-Programmierung weitestgehend verborgen. Entfernte Methoden werden prinzipiell wie lokale Methoden aufgerufen. Mit Hilfe der Objektserialisierung kann RMI beliebige Objekte als Aufrufparameter entfernter Methoden übertragen. Dabei kann auch der zugehörige Bytecode, sofern er lokal nicht vorhanden ist, ad hoc transferiert werden. Damit sind dann sehr flexible Anwendungen möglich: mobile Agenten, Anwendungen mit automatischem Rückruf (Callback) bei Eintritt bestimmter Ereignisse. Das Kapitel schließt mit einem Exkurs, der zeigt, wie RMI mit Hilfe des CORBA-Protokolls *IIOP* CORBA-fähig gemacht werden kann.

8 Nachrichtendienste mit JMS
Ein wichtiger Standard für die asynchrone Verteilung von Nachrichten auf der Basis nachrichtenorientierter Middleware ist der *Java Message Service* (JMS). Bekannte Kommunikationsmodelle, um Nachrichten auszutauschen sind: das *Point-to-Point-Modell* und das *Publish-Subscribe-Modell*. In diesem Kapitel werden die wichtigsten Schnittstellen des JMS-API vorgestellt.

Erwartungen an den Leser

Von den Leserinnen und Lesern werden erwartet:

- Grundlegende Java-Kenntnisse, insbesondere: Konzepte der objektorientierten Programmierung, Dateiverarbeitung und Thread-Programmierung.
- Grundlegende Kenntnisse auf dem Gebiet der relationalen Datenbanken und SQL.
- Das Wissen um Internet und World Wide Web und der Umgang mit einem Webbrowser.
- Grundlagenkenntnisse in HTML und XML.

Die Beispiele

Alle Beispielprogramme und Lösungen zu den Übungsaufgaben wurden mit Hilfe von Java SE 5.0 und Java SE 6.0 unter Windows 2000 und Windows XP entwickelt und im Netz getestet. Selbstverständlich sind die hier vorgestellten Programme ohne Änderung auch auf UNIX/Linux-Systemen lauffähig.

Werkzeug Ant

Zur Vereinfachung der Programmentwicklung wurde das Build-Werkzeug *Ant* der Apache Software Foundation genutzt (siehe Quellenverzeichnis am Ende des Buches). Im Anhang befindet sich eine kurze Anleitung zur Installation und Nutzung von *Ant* für die Programme dieses Buches.

Weitere Einzelheiten, z.B. zu den eingesetzten Datenbanksystemen und JDBC-Treibern, können Kapitel 2 entnommen werden. Bezugsquellen können am Ende des Buches nachgeschlagen werden.

Download

Sämtliche Programme und Lösungen stehen im Internet zur Verfügung und können heruntergeladen werden. Hinweise zu weiteren Tools und Klassenbibliotheken erfolgen in den Kapiteln, in denen sie erstmalig benutzt werden. Das Quellenverzeichnis am Ende des Buches enthält die Bezugsquellen.

1.2 Verteilte Systeme

Umfassende gesellschaftliche und wirtschaftliche Entwicklungen zwingen Unternehmen, sich flexibel an Veränderungen anzupassen und schnell auf neue Marktsituationen zu reagieren. Betrieblichen Anwendungssystemen kommt hier eine besondere Bedeutung zu. Wie sind die Anwendungssysteme zu implementieren, um höchstmögliche Flexibilität, Verlässlichkeit und Anpassbarkeit zu erreichen?

Monolithische Systeme (alle Anwendungen laufen auf *einem* Rechner) sind nur noch bedingt geeignet. Heute ist die Strukturierung der Software in *Client* und *Server* weit verbreitet.

Die folgenden Faktoren haben die fundamentalen Änderungen wesentlich beeinflusst:

- Leistungsexplosion in der Mikroprozessortechnik,
- schnelle Datennetze,
- Fortschritte in der Softwaretechnik,
- Abkehr von hierarchischen Organisationsstrukturen,
- Bedürfnis nach Integration bisher nicht integrierter Systeme,
- Internet-Technologie.

Waren die 1960er Jahre noch durch *Batch-Processing-Systeme*, Lochstreifen und Lochkarten geprägt, konnten Anfang der 1970er Jahre in so genannten *Timesharing-Systemen* Transaktionen im Dialog gestartet werden. Minicomputer (UNIX-Systeme) erschienen als zweiter Gerätetyp neben dem Host. Anfang der 1980er

Jahre kamen die *Personal Computer* als Stand-alone-Systeme oder vernetzt im LAN hinzu. Der wichtigste Trend ab 1990 ist durch das Aufkommen von *Client/Server-Systemen* für die verteilte Verarbeitung bestimmt. Hierzu zählen insbesondere die heutigen Internet-Anwendungen (Web-Informationssysteme, E-Business-Anwendungen).

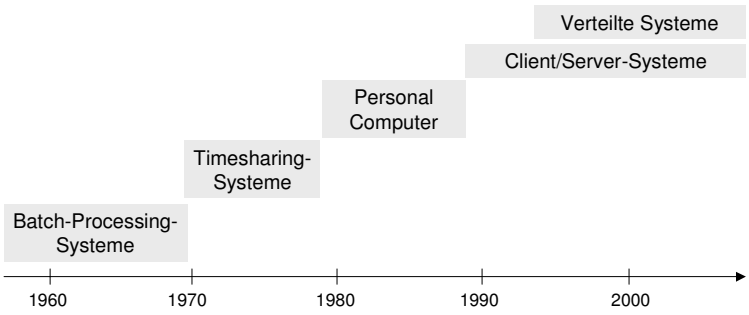


***Bild 1.1**: Entwicklungsstufen*

Jede Anwendung kann in drei wesentliche Schichten eingeteilt werden:

1. *Präsentation*
   Diese Schicht (auch Benutzungsschnittstelle genannt) hat die Aufgabe, den Dialog mit dem Benutzer zu steuern sowie Daten und Befehle an das System zu übermitteln.
2. *Verarbeitung*
   Diese Schicht stellt die Verarbeitungslogik dar. Hier wird der eigentliche Nutzen der Anwendung erzielt.
3. *Datenhaltung*
   Diese Schicht hat die Aufgabe, die Daten abzuspeichern und bei Bedarf wieder zur Verfügung zu stellen.

Bild 1.2 zeigt eine Webanwendung, die dem Benutzer das Produktangebot eines Unternehmens präsentiert. Die drei Schichten Präsentation, Verarbeitung und Datenhaltung sind jeweils auf eigenen Rechnern implementiert. Der *Webbrowser* ist die Benutzungsoberfläche, die die abgefragten Informationen grafisch präsentiert. Der *Webserver* erzeugt Datenbankabfragen, bereitet die Abfrageergebnisse als Webseite auf und übermittelt diese an den Client. Der *Datenbankserver* verwaltet den Produktkatalog und führt die Datenbankabfragen aus. Webbrowser, Webserver und Datenbankserver sind autonome Teilsysteme, die koordiniert miteinander kooperieren.

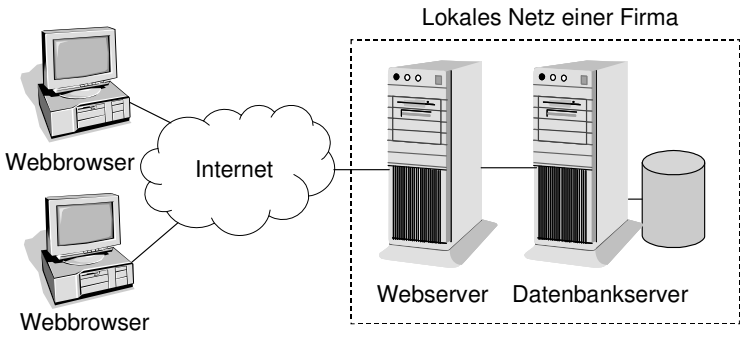


***Bild 1.2**: Beispiel einer Webanwendung*

Verteiltes System

Verteilung bedeutet die Zuordnung von Funktionen und Daten auf mehrere Rechner. Ein Anwendungssystem wird dabei in funktionale Komponenten zerlegt, die dann bei der Installation im Netz so verteilt werden, dass die Komponenten optimal unterstützt werden.

Abstrakt formuliert ist ein *verteiltes System* eine Menge von Komponenten, die kooperieren, um eine *gemeinsame* Aufgabe zu erfüllen.

Konfigurationsproblem

Da es im Allgemeinen mehrere Varianten für die Installation der Komponenten im Netz gibt, besteht das *Konfigurationsproblem*, die *logische* Struktur der Anwendung auf die *physische* Struktur (Rechner im Netz) so abzubilden, dass bei vorgegebenen Randbedingungen ein maximaler Nutzen erzielt wird.

Ob eine Verteilung überhaupt möglich ist, entscheidet sich beim Design der Anwendungssoftware. Das Design muss unabhängig von einer späteren physischen Konfiguration sein. Dem Benutzer muss die konkrete physische Aufteilung verborgen bleiben.

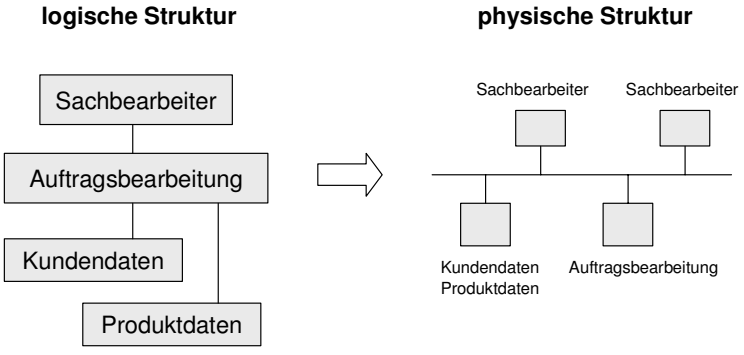


***Bild 1.3**: Abbildung der logischen auf die physische Struktur*

Der Vergleich der verteilten Verarbeitung mit der großrechnerdominierten, zentralen Verarbeitung ergibt folgende Vor- und Nachteile:

Vorteile

- *Besseres Abbild der Realität*
  Die Verteilung unterstützt die "schlanke" Organisation. Leistungen werden dort erbracht, wo sie benötigt werden. Daten werden dort erfasst, wo sie im Geschäftsprozess entstehen.
- *Wirtschaftlichkeit*
  Teure Ressourcen (Geräte, Softwarekomponenten) stehen mehreren Rechnern zur Verfügung und können gemeinsam genutzt werden. Automatisierte Aufgaben werden möglichst dort ausgeführt, wo sie am wirtschaftlichsten sind.
- *Bessere Lastverteilung*
  Da mehrere Rechner mit jeweils eigenem Betriebssystem arbeiten, kann durch teilweise parallele Verarbeitung die Gesamtbearbeitungszeit verkürzt werden.
- *Bessere Skalierbarkeit*
  Einzelne Komponenten können leichter an einen steigenden Bedarf angepasst werden.
- *Fehlertoleranz*
  Beim Ausfall eines Rechners bleiben die anderen betriebsbereit. Die Aufgaben des ausgefallenen Rechners können oft von einem anderen Rechner übernommen werden.

Nachteile

- *Höhere Komplexität durch Verteilung und Heterogenität*
  Die Verteilung der Komponenten großer Systeme und die Heterogenität bei Betriebssystemen, Programmiersprachen, Datenformaten und Kommunikationsprotokollen stellen hohe Anforderungen an die Entwickler, wenn das Gesamtsystem überschaubar bleiben soll.
- *Komplexe Netzinfrastrukturen*
  Netzarchitektur und Netzmanagement sind das Rückgrat der Systeme eines Unternehmens. Die Integration heterogener Systeme mit Komponenten unterschiedlicher Hersteller- und Standardarchitekturen stellt hohe Anforderungen an die Administration.
- *Höhere Sicherheitsrisiken*
  Verteilte Systeme bieten mehr Möglichkeiten für unberechtigte Zugriffe. Im Netz übertragene Nachrichten können abgehört oder sogar verändert werden. Der Einsatz von Verschlüsselungsverfahren und Firewall-Systemen sind wirksame Schutzmaßnahmen.

Aufgabe von so genannten *Verteilungsplattformen* ist es, diese Komplexität beherrschbar zu machen.

1.3 Das Client/Server-Modell

Das *Client/Server-Modell* ist das am weitesten verbreitete Modell für die verteilte Verarbeitung.

Client/Server

Client/Server bezeichnet die Beziehung, in der zwei Programme zueinander stehen. Der *Client* stellt eine Anfrage an den Server, eine gegebene Aufgabe zu erledigen. Der *Server* erledigt die Aufgabe und liefert das Ergebnis beim Client ab. Ein Kommunikationsdienst verbindet Client und Server miteinander.

Ein *Client/Server-System* besteht also aus

- einem oder mehreren Clients (Auftraggebern),
- einem Server (Auftragnehmer) und
- einem Kommunikationsdienst (Netzwerk, Vermittler).

Eine Softwarearchitektur

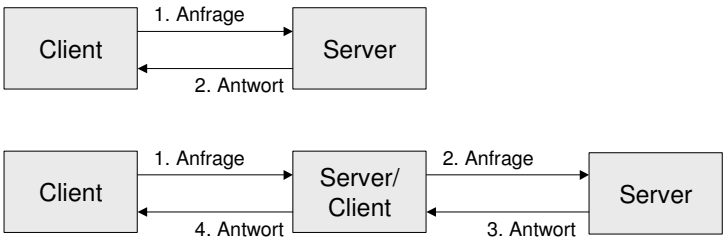
Client und Server sind Programme, die auch auf demselben Rechner laufen können. Sie müssen also nicht notwendig über ein Netz verbunden sein. Das macht deutlich, dass es sich hierbei um eine Software- und keine Hardwarearchitektur handelt. In der Praxis werden natürlich Client und Server auf unterschiedlichen Rechnern verteilt, um die bekannten Vorteile eines verteilten Systems zu erzielen.

Hardware-Sicht

Gelegentlich werden auch die *Trägersysteme* (Rechner), auf denen die Programme laufen, als Client bzw. Server bezeichnet. Wir nutzen die Begriffe Client bzw. Server als Homonyme. Aus dem Zusammenhang wird dann klar, welche Bedeutung gemeint ist.

Ein Server kann die Erledigung einer Aufgabe weiter delegieren und dazu auf andere Server zugreifen. Er spielt dann selbst die Rolle eines Client. Der Webserver in Bild 1.2 hat diese Doppelfunktion.

***Bild 1.4**: Client/Server-Modell*



Im Folgenden sind charakteristische Merkmale von Client und Server zusammengefasst:

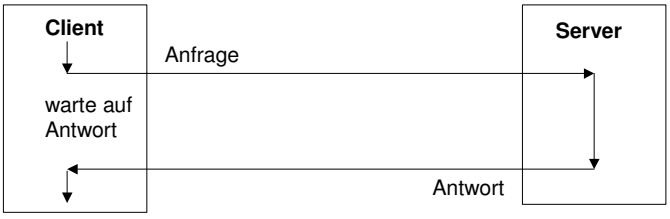
Client-Merkmale

- Ein Programm wird vorübergehend zum Client, wenn es einen Dienst von einem anderen Programm (Server) anfordert. Darüber hinaus kann das Programm andere Aufgaben lokal ausführen.
- Der Client läuft in der Regel auf dem Rechner eines Benutzers und leitet den Kontakt mit einem Server *aktiv* ein.
- Der Client kann im Laufe einer Sitzung auf mehrere Server zugreifen, kontaktiert aber aktiv nur jeweils einen Server.

Server-Merkmale

- Der Server ist ein Programm, das einen bestimmten Dienst bereitstellt. Er kann in der Regel gleichzeitig mehrere Clients bedienen.
- Häufig werden Server automatisch beim Hochfahren des Rechners gestartet und beim Herunterfahren beendet.
- Der Server wartet *passiv* auf die Verbindungsaufnahme durch einen Client.
- Da Server eigenständige Programme sind, können mehrere Server unabhängig voneinander auf einem einzigen Rechner als Trägersystem laufen.
- *Parallelbetrieb* ist ein grundlegendes Merkmal von Servern. Mehrere Clients können einen bestimmten Dienst in Anspruch nehmen, ohne warten zu müssen, bis der Server mit der Erledigung der laufenden Anfrage fertig ist. In den späteren Java-Programmbeispielen werden die Client-Anfragen jeweils in einem neuen Thread bedient.

synchrone Kommunikation



asynchrone Kommunikation

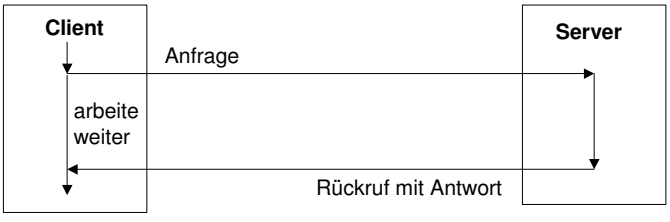


***Bild 1.5**: Interaktionsarten*

synchron/asynchron

Die Interaktion zwischen Client und Server kann *synchron* oder *asynchron* erfolgen (siehe Bild 1.5).

Bei der *synchronen* Kommunikation wartet der Client nach Absenden der Anfrage an den Server so lange, bis er eine Rückantwort erhält, und kann dann erst andere Aktivitäten ausführen.

Im *asynchronen* Fall sendet der Client die Anfrage an den Server und arbeitet sofort weiter. Beim Eintreffen der Rückantwort werden dann bestimmte Ereignisbehandlungsroutinen beim Client aufgerufen. Die durch den Server initiierten Rückrufe (*Callbacks*) erfordern allerdings zusätzliche Kontrolle beim Client, wenn ungewünschte Unterbrechungen der Arbeit vermieden werden sollen.

1.4 Mehrstufige Architekturen

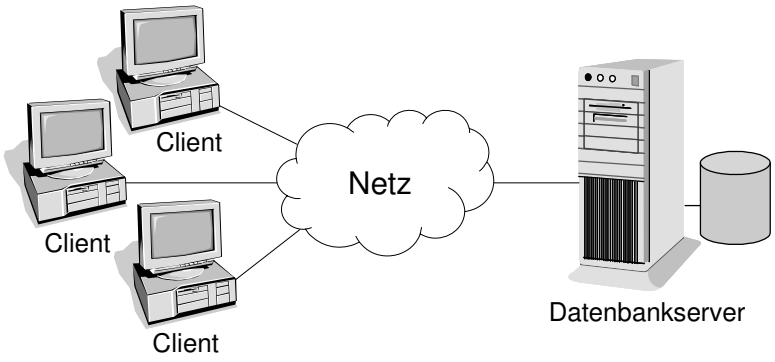
Bei *einstufigen Architekturen* wird die gesamte Anwendung (Präsentation, Verarbeitung und Datenhaltung) auf *einem* Rechner implementiert. Die Benutzerinteraktion erfolgt über Terminals, die direkt oder über einen Terminal- bzw. Kommunikationsserver an den Rechner angeschlossen sind.

Multi-Tier-Architekturen

Verteilte Anwendungen haben eine *mehrstufige Architektur* (*Multi-Tier-Architektur*). Beispielsweise werden Applikations- und Datenbankserver verschiedenen Ebenen zugeordnet.

Bei der *zweistufigen Architektur* ist die Gesamtanwendung zwei Ebenen zugeordnet: Client und Server.

***Bild 1.6**: Eine zweistufige Architektur*



Für die Arbeitsteilung zwischen Client und Server existieren verschiedene Alternativen, je nachdem, wo die Schichten *Präsentation*, *Verarbeitung* und *Datenhaltung* angesiedelt sind.

Bild 1.7 zeigt fünf Alternativen der Aufgabenverteilung bei der zweistufigen Architektur eines Client/Server-Systems. Von Alternative 1 bis 5 wird immer mehr Funktionalität auf den Client übertragen.

Alternative 1 zeigt den Fall einer Webanwendung, bei der HTML-Seiten vom Server generiert und vom Webbrowser angezeigt werden.

Ein Beispiel zur Alternative 2 ist eine Webanwendung, bei der eingegebene Daten über ein Applet im Webbrowser an den Server zur Verarbeitung weitergeleitet werden.

Bei den Alternativen 3 und 5 hat der Client einen Teil der Verarbeitung bzw. Datenhaltung selbst übernommen.

Alternative 4 zeigt den Fall eines typischen Datenbankservers.

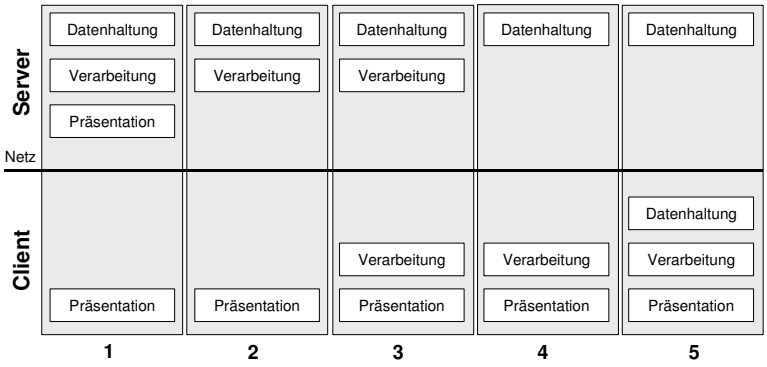


***Bild 1.7**: Alternativen der Aufgabenverteilung*

Bei der *dreistufigen Architektur* wird die Gesamtanwendung auf drei Ebenen aufgeteilt: z.B. Client, Applikationsserver und Datenbankserver. Dieses Modell ist auch Basis vieler ERP-Systeme (Enterprise Resource Planning).

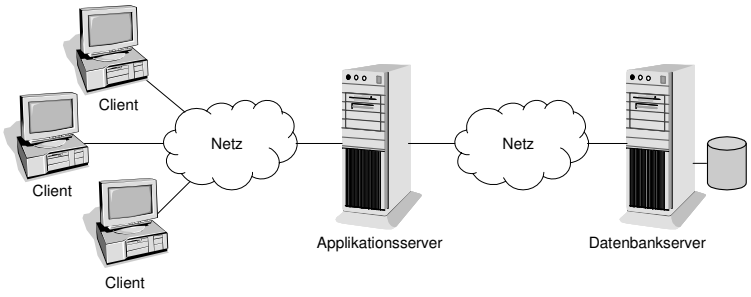


***Bild 1.8**: Eine dreistufige Architektur*

Insbesondere bei Internet- und Intranet-Anwendungen findet man eine *vierstufige Architektur*, bei der ein Webserver dem Applikationsserver vorgeschaltet ist.

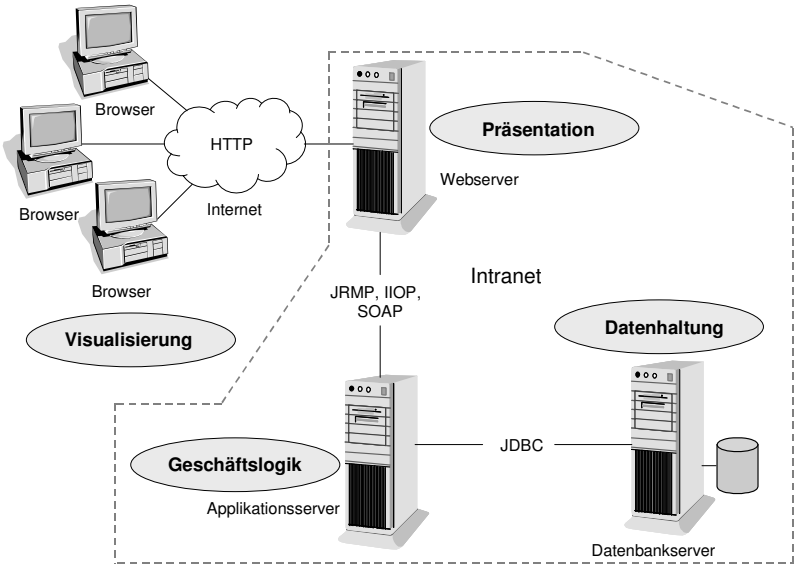


***Bild 1.9**: Eine vierstufige Architektur*

1.5 Middleware und Transparenz

In mehrstufigen Architekturen sind die Komponenten eines Anwendungssystems auf potentiell heterogene Trägersysteme in einem Netzwerk verteilt. Um die Komplexität des Gesamtsystems bei der Bedienung oder Entwicklung der Anwendung vor dem Benutzer bzw. Entwickler verborgen zu halten, bedarf es einer geeigneten Software-Infrastruktur, die die Interaktion zwischen den Komponenten in besonderer Weise unterstützt.

Die verteilte Verarbeitung hat zu einem neuen Typ systemnaher Software geführt, der so genannten *Middleware*.

Middleware ist Kommunikationssoftware, die den Austausch von Informationen zwischen den verschiedenen Komponenten eines verteilten, heterogenen Systems unterstützt. Die Anwendungen werden dabei von den komplexen Details der internen Vorgänge abgeschirmt.

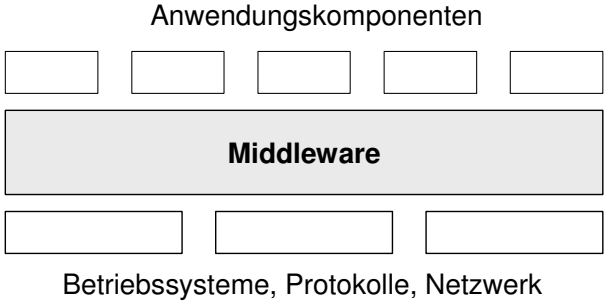


***Bild 1.10**: Middleware*

In der Informatik nennt man eine Sache *transparent*, wenn sie "durchsichtig", also nicht sichtbar ist.

Verteilung verbergen

In verteilten Systemen sollen interne Abläufe und Implementierungsdetails für den Benutzer oder Anwendungsentwickler nicht sichtbar sein. *Verteilungstransparenz* verbirgt die Komplexität verteilter Systeme.

Es existieren mehrere untergeordnete Transparenzbegriffe, die jeweils einen bestimmten Aspekt bezeichnen. Beispiele hierfür sind:

- *Ortstransparenz*
  Der Ort einer Ressource ist dem Benutzer nicht bekannt. Er identifiziert sie über einen Namen, der keine Information über ihren Aufenthaltsort enthält.
- *Zugriffstransparenz*
  Die Form des Zugriffs auf eine Ressource ist einheitlich und unabhängig davon, ob die Ressource lokal oder auf einem entfernten Rechner zur Verfügung steht. Unterschiede verschiedener Betriebssysteme und Dateisysteme werden verborgen.
- *Nebenläufigkeitstransparenz*
  Der gleichzeitige Zugriff mehrerer Benutzer auf dieselbe Ressource (z.B. Datenbanktabelle) erfolgt ohne gegenseitige Beeinflussung und lässt keine falschen Ergebnisse entstehen.
- *Replikationstransparenz*
  Sind aus Verfügbarkeits- oder Performance-Gründen mehrere Kopien einer Ressource (z.B. eines Datenbestandes) vorhanden, so merkt der Benutzer nicht, ob er auf das Original oder eine Kopie zugreift. Das System sorgt dafür, dass alle Kopien bei Änderungen konsistent bleiben.
- *Migrationstransparenz*
  Ressourcen können von einem Rechner auf einen anderen verlagert werden, ohne dass der Benutzer dies bemerkt.
- *Fehlertransparenz*
  Der Benutzer soll nicht mit allen auftretenden Fehlern im System konfrontiert werden. Das Auftreten von Fehlern und die Fehlerbehebung sollen vor dem Benutzer weitestgehend verborgen sein.

Middleware-Produkte stellen dem Entwickler die für die Anwendung benötigten Funktionen meist in Form eines *API* (Application Programming Interface) zur Verfügung.

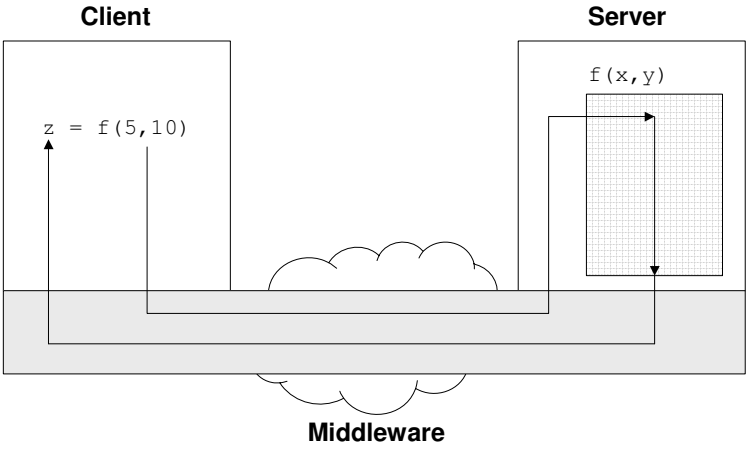
Datenbank-Middleware

Datenbank-Middleware ermöglicht den Zugriff auf unterschiedliche Datenbanksysteme, ohne das Anwendungsprogramm beim Wechsel des Datenbanksystems ändern zu müssen. Das *JDBC-*

API erlaubt es einem Java-Programm, SQL-Anweisungen an beliebige SQL-Datenbanken zu schicken (siehe Kapitel 2).

Andere Middleware-Produkte verwenden eine synchrone Verbindung zwischen Client und Server, um Prozeduren nach dem *RPC-Modell* aufzurufen.

***Bild 1.11**: Entfernter Prozeduraufruf*



RPC

Der *Remote Procedure Call* (*RPC*) versteckt den Aufruf einer auf einem anderen Rechner im Netz implementierten Prozedur hinter einem lokalen Prozeduraufruf und bietet damit ein sehr vertrautes Programmiermodell an. Das Programm, das die Prozedur aufruft, agiert als Client, das Programm, das die aufgerufene Prozedur ausführt, als Server.

RMI (*Remote Method Invocation*) ist die objektorientierte Umsetzung des RPC-Modells für Java-Programme (siehe Kapitel 7).

Die folgenden Programmbeispiele zeigen, wie einfach eine Anwendung mit *lokalem Methodenaufruf* in eine Client/Server-Anwendung mit *entferntem Methodenaufruf* (RMI) umgewandelt werden kann.

Die im Beispiel benutze Methode berechnet die Summe zweier Zahlen.

Zunächst die lokale Anwendung:

Programm 1.1

```
public class AddServer {
  public double add(double x, double y) {
    return x + y;
  }
}

public class AddClient {
  public static void main(String[] args) {
    AddServer service = new AddServer();
    System.out.println(service.add(2.3, 5.7));
  }
}
```

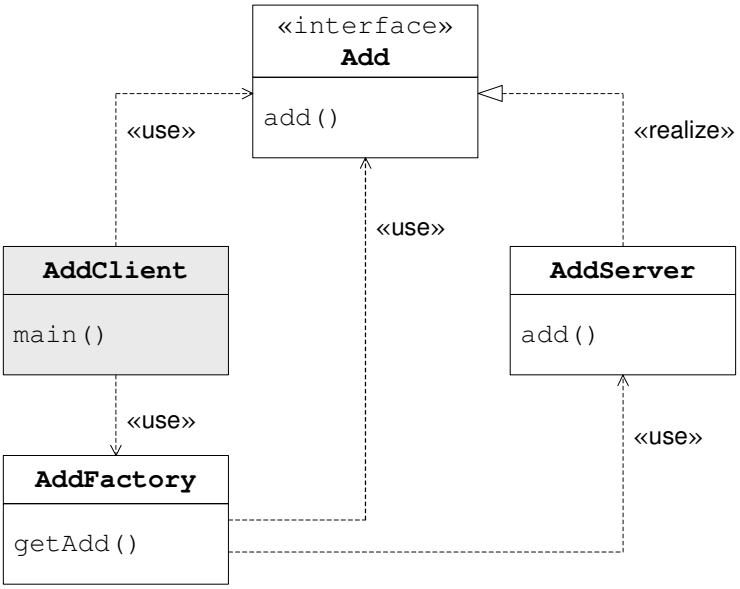
Die beiden Klassen werden nun mittels eines Interface entkoppelt. Um ein Server-Objekt zu erzeugen, wird eine statische Methode benutzt, die den konkreten Namen der Implementierungsklasse (hier: `AddServer`) gegenüber dem Client verbirgt (*Factory-Methode*).

Programm 1.2

```
public interface Add {
  double add(double x, double y);
}

public class AddServer implements Add {
  public double add(double x, double y) {
    return x + y;
  }
}

public class AddFactory {
  public static Add getAdd() {
    return new AddServer();
  }
}

public class AddClient {
  public static void main(String[] args) {
    Add service = AddFactory.getAdd();
    System.out.println(service.add(2.3, 5.7));
  }
}
```

Das Klassendiagramm in Bild 1.12 verdeutlicht die Zusammenhänge.

Bild 1.12: *Klassendiagramm zu Programm 1.2*



Die RMI-Version der Anwendung entsteht nun durch leichte Änderung von Programm 1.2.

Programm 1.3

```
import java.rmi.*;

public interface Add extends Remote {
  double add(double x, double y) throws RemoteException;
}
```

```
import java.rmi.*;
import java.rmi.server.*;

public class AddServer extends UnicastRemoteObject implements Add {
  public AddServer() throws RemoteException { }

  public double add(double x, double y) throws RemoteException {
    return x + y;
  }

  public static void main(String[] args) throws Exception {
    AddServer server = new AddServer();
    Naming.rebind("add", server);
  }
}
```

```
import java.rmi.*;

public class AddFactory {
  public static Add getAdd() throws Exception {
    return (Add) Naming.lookup("//localhost/add");
  }
}


public class AddClient {
  public static void main(String[] args) throws Exception {
    Add service = AddFactory.getAdd();
    System.out.println(service.add(2.3, 5.7));
  }
}
```

`Add`, `AddServer` und `AddFactory` importieren RMI-Pakete.

Das Interface `Add` ist von `java.rmi.Remote` abgeleitet, die Methode `add` enthält eine `throws`-Klausel mit der Ausnahme `java.rmi.RemoteException`.

Die Klasse `AddServer` ist von `java.rmi.server.UnicastRemoteObject` abgeleitet und besitzt eine `main`-Methode, die das erzeugte Server-Objekt bei einem *Verzeichnisdienst* registriert.

Die Methode `getAdd` der Klasse `AddFactory` erhält ihr `Add`-Objekt mit Hilfe dieses Verzeichnisdienstes.

Zur Vereinfachung sollen Client und Server auf demselben Rechner laufen.

Zunächst müssen die Sourcen compiliert werden:

```
mkdir build

javac -sourcepath src -d build src/*.java
```

Start des Verzeichnisdienstes:

```
start /Dbuild rmiregistry
```

Start des Servers:

```
start java -cp build AddServer
```

Aufruf des Client:

```
java -cp build AddClient
```

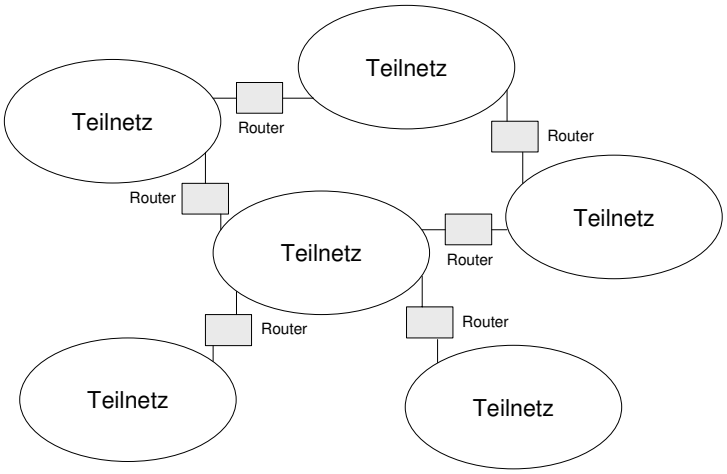
1.6 Grundbegriffe des Internets

Internet

Das *Internet* ist ein weltumspannendes Rechnernetz, das aus einer Vielzahl von internationalen und nationalen öffentlichen Netzen sowie privaten lokalen Netzen besteht. Spezielle Kopplungselemente, so genannte *Router*, verbinden diese Teilnetze miteinander und ermöglichen so den Datenaustausch zwischen Rechnern, die sich in unterschiedlichen Teilnetzen befinden. Das Internet ist ein offenes Netz, zu dem Unternehmen, nichtkommerzielle Organisationen sowie Privatpersonen gleichermaßen Zugang haben.

Kommunikationsprotokolle beinhalten Sprach- und Handlungsregeln für den Datenaustausch. Sie sind das Verständigungsmittel in einem Rechnernetz.

***Bild 1.13**: Grundstruktur des Internets*



TCP/IP

TCP/IP (Transmission Control Protocol/Internet Protocol) bezeichnet eine Familie von einzelnen, aufeinander abgestimmten Protokollen für die Kommunikation im Internet. TCP und IP sind die beiden wichtigsten Protokolle dieser Familie.

Intranet

Im Gegensatz zum offenen Internet ist ein *Intranet* ein privates, innerbetriebliches Rechnernetz auf Basis der Internet-Protokolle ("privates Internet").

Zur Beschreibung der Kommunikation werden allgemein *Schichtenmodelle* eingesetzt. Im Internet werden vier aufeinander aufbauende Schichten unterschieden. Jede Schicht hat ihre eigene Funktionalität, die sie der jeweils höheren Schicht zur Verfügung stellt.

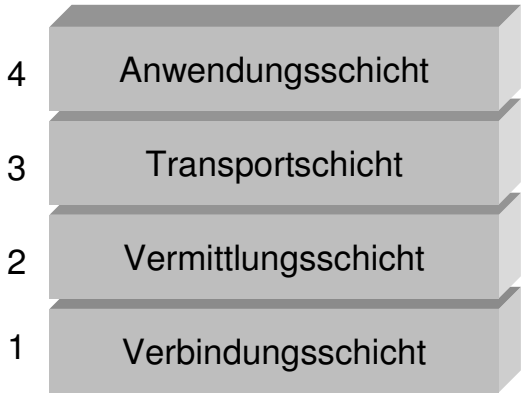


***Bild 1.14**: TCP/IP-Schichtenmodell*

1. Die *Verbindungsschicht* umfasst die Netzwerkhardware und Gerätetreiber. Sie ist die Basis für eine zuverlässige physische Verbindung zwischen zwei benachbarten Systemen.
2. Die *Vermittlungsschicht* enthält das Protokoll *IP* (Internet Protocol), das die Internet-Adressen (IP-Adressen) definiert, die als Basis für die Wegwahl im Internet dienen. Einzelne Datenpakete werden vom Sender zum Empfänger über verschiedene Teilnetze des Internets geleitet. Das hier angesiedelte Protokoll *ARP* (Address Resolution Protocol) übersetzt logische IP-Adressen in physische Adressen (Hardwareadressen) einer Datenstation im lokalen Netz.
3. Die Transportschicht enthält die beiden Protokolle *TCP* (Transmission Control Protocol) und *UDP* (User Datagram Protocol).

   TCP stellt der Anwendung ein verlässliches, *verbindungsorientiertes* Protokoll zur Verfügung. Vor der eigentlichen Datenübertragung in Form von Datenpaketen wird eine virtuelle Ende-zu-Ende-Verbindung zwischen Sender und Empfänger aufgebaut.

   UDP ist ein so genanntes *verbindungsloses* Protokoll. Datenpakete werden ins Internet geschickt, ohne dass vorher eine Verbindung mit dem Empfänger hergestellt wurde.

   UDP und TCP werden in den Kapiteln 3 und 4 ausführlich behandelt.
4. Beispiele für die Protokolle der *Anwendungsschicht* sind: *SMTP* (Simple Mail Transfer Protocol), *Telnet*, *FTP* (File Transfer Protocol) und *HTTP* (Hypertext Transfer Protocol), das die Basis für das *World Wide Web* (WWW) ist.

Bild 1.15 zeigt den Ablauf einer Kommunikation zwischen Client und Server, die ihre Daten direkt über TCP austauschen.

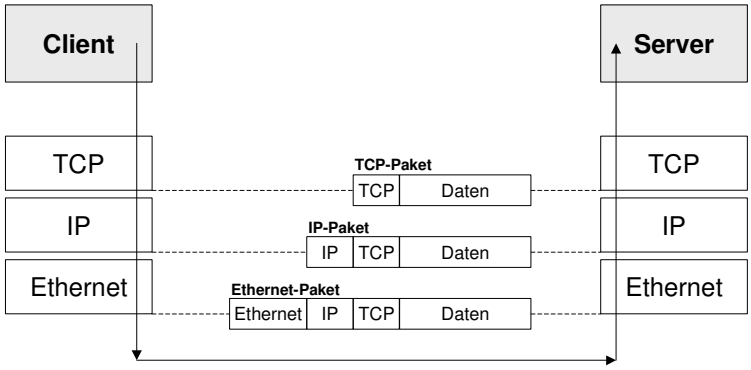


***Bild 1.15**: Datenübertragung mittels TCP*

Der Client übergibt seine Daten an die Transportschicht. TCP sorgt für die Aufteilung der Daten in Pakete, versieht jeweils die Nutzdaten eines Pakets mit einem Datenkopf (Header) und gibt die Pakete weiter an die Vermittlungsschicht. Die Vermittlungsschicht fügt ihrerseits einen IP-Header hinzu. Schließlich packt die Netzwerkkarte einen Rahmen um die Daten. Beim Empfänger werden in umgekehrter Reihenfolge die Header Schicht für Schicht wieder entfernt und ausgewertet, bevor die Daten an die übergeordnete Schicht weitergegeben werden.

IP-Adressen

Eine Aufgabe des Protokolls IP ist die Bereitstellung eines Adressierungsschemas, das von den physischen Adressen der Netzwerkhardware unabhängig ist.

Jeder Rechner im Internet (genauer: der Rechner als Komponente eines Teilnetzes) hat eine eindeutige *IP-Adresse*, die in der zur Zeit noch vorherrschenden Version IPv4 aus 32 Bit besteht und durch eine Folge von vier durch Punkte getrennte Zahlen zwischen 0 und 255 dargestellt wird.

Eine IP-Adresse besteht aus zwei Teilen: der *Netzadresse*, die das Netz eindeutig identifiziert, und der *Rechneradresse*, die den Rechner in diesem Netz eindeutig identifiziert. Offizielle Netzadressen werden zentral vergeben, Rechneradressen können frei vom Netzwerkadministrator der jeweiligen Institution vergeben werden.

Drei *Adressklassen* werden unterschieden: A, B und C. Adressen der Klasse A nutzen 8 Bit für die Netzadresse (beginnend mit 0), Adressen der Klasse B 16 Bit (beginnend mit 10) und Adressen der Klasse C 24 Bit (beginnend mit 110).

Beispiel:

Die IP-Adresse `194.94.124.236` gehört zur Klasse C und identifiziert einen Rechner in dem durch die Netzadresse `194.94.124.0` identifizierten Netz.

Jeder Rechner im Internet kann sich selbst mit der Adresse `127.0.0.1` adressieren.

Private IP-Adressen wie z.B. `10.0.0.0` bis `10.255.255.255` oder `192.168.0.0` bis `192.168.255.255` werden nie im Internet geroutet. Jede Organisation kann sie frei im internen Netz verwenden.

Die starre Klasseneinteilung der IP-Adressen wird durch das heute übliche Verfahren *CIDR* (Classless Inter Domain Routing) aufgehoben. Hierbei erfolgt die Aufteilung in Netz- und Rechneradresse bitweise mit Hilfe von Subnetzmasken.

DNS

Zur besseren Handhabbarkeit wird das Nummernsystem durch ein Namensystem, dem *Domain Name System* (DNS), überlagert. IP-Adressen werden einem sprechenden Namen, dem *Domain-Namen*, zugeordnet. Domain-Namen sind hierarchisch aufgebaut.

Beispiel: `www.hs-niederrhein.de`

Hier handelt es sich um den Webserver der Hochschule Niederrhein mit `de` (Deutschland) als *Top Level Domain*.

Der Hostname `localhost` identifiziert den eigenen Rechner und entspricht der IP-Adresse `127.0.0.1`.

DNS-Server sind spezielle Verzeichnisdienste, die die Zuordnung von IP-Adressen zu DNS-Namen und umgekehrt unterstützen. Hierüber kann z.B. die IP-Adresse zu einem Namen erfragt werden.

Zu beachten ist, dass mehrere Rechner mit jeweils eigener IP-Adresse denselben DNS-Namen haben können und zu einer IP-Adresse mehrere DNS-Namen gehören können.

Portnummer

Um einen bestimmten Dienst (Server) im Internet zu identifizieren, reicht die IP-Adresse des Rechners, auf dem der Server läuft, nicht aus, da mehrere Server auf demselben Rechner gleichzeitig laufen können.

Zur Identifizierung eines Servers dient neben der IP-Adresse des Rechners die so genannte *Portnummer*. Portnummern werden auch vom Server benutzt, um den anfragenden Client für die Rückantwort zu adressieren. Portnummern sind ganze Zahlen von 0 bis 65535. Sie werden von den Protokollen der Transportschicht verwendet. TCP und UDP können Portnummern unabhängig voneinander verwenden, d.h. ein TCP-Server und ein UDP-Server können auf einem Rechner unter derselben Port-

nummer zur gleichen Zeit laufen. Die Portnummern von 0 bis 1023 sind weltweit eindeutig definiert und für so genannte *well-known services* reserviert.

***Bild 1.16**: Beispiel für well-known Services*

Port	Protokoll	Service	Beschreibung
7	TCP/IP	echo	liefert die Eingabe als Antwort zurück
13	TCP/UDP	daytime	liefert die aktuelle Zeit des Servers
20	TCP	ftp-data	überträgt Daten zum Server
21	TCP	FTP	sendet FTP-Kommandos zum Server
23	TCP	Telnet	Terminalemulation
25	TCP	SMTP	überträgt E-Mails zwischen Rechnern
53	TCP/UDP	DNS	Domain Name System
80	TCP	HTTP	Webserver

Die Liste der *well-known* Portnummern kann über

```
http://www.iana.org/assignments/port-numbers
```

abgefragt werden.

Im Bereich von 1024 bis 49151 sind weitere Portnummern registriert, z.B. 1099 für die RMI-Registry und 3306 für den MySQL-Datenbankserver.

1.7 Die Klasse InetAddress

Die Klasse `java.net.InetAddress` repräsentiert eine IP-Adresse. Subklassen hiervon sind `Inet4Address` und `Inet6Address`, die IP-Adressen der Version 4 bzw. 6 speichern können.

Die folgenden drei statischen Methoden erzeugen `InetAddress`-Objekte:

```
static InetAddress getLocalHost() throws UnknownHostException
```

ermittelt die IP-Adresse des Rechners, auf dem diese Methode läuft.

Die Klasse `java.net.UnknownHostException` ist Subklasse von `java.io.IOException`. Eine Ausnahme dieses Typs wird ausgelöst, wenn die IP-Adresse eines Rechners nicht ermittelt werden kann.

```
static InetAddress getByName(String host) throws UnknownHostException
```

liefert die IP-Adresse zu einem Hostnamen. Enthält das Argument `host` eine IP-Adresse in Punkt-Notation anstelle eines Namens, so wird nur die Gültigkeit des Adressformats geprüft.

`static InetAddress[] **getAllByName**(String host)`
`    throws UnknownHostException`

liefert ein Array von IP-Adressen, die alle zum vorgegebenen Hostnamen gehören. Ergebnisse früherer Adressauflösungen innerhalb eines Programmlaufs werden zwischengespeichert und wiederverwendet.

`String **getHostAddress**()`

liefert die IP-Adresse eines `InetAddress`-Objekts in Punkt-Notation.

`String **getHostName**()`

liefert den Hostnamen der IP-Adresse. Falls das `InetAddress`-Objekt mittels einer IP-Adresse in Punkt-Notation erzeugt wurde und der zugehörige Hostname nicht ermittelt werden konnte, wird die IP-Adresse in Punkt-Notation zurückgegeben.

Programm 1.4 zeigt die IP-Adresse und den Hostnamen des eigenen Rechners an.

Programm 1.4

```
import java.net.*;

public class LocalHost {
  public static void main(String[] args) throws UnknownHostException {
    InetAddress addr = InetAddress.getLocalHost();
    System.out.println("IP-Nummer:\t" + addr.getHostAddress());
    System.out.println("Hostname:\t" + addr.getHostName());
  }
}
```

Programm 1.5 liefert die IP-Adresse zu einem Hostnamen.

Programm 1.5

```
import java.net.*;

public class Lookup {
  public static void main(String[] args) throws UnknownHostException {
    if (args.length != 1) {
      System.err.println("java Lookup (<hostname> | <ip-adresse>)");
      System.exit(1);
    }

    InetAddress addr = InetAddress.getByName(args[0]);
    System.out.println("IP-Nummer:\t" + addr.getHostAddress());
    System.out.println("Hostname:\t" + addr.getHostName());
  }
}
```

Test

Wird im lokalen Netz kein Domain Name Server (DSN) genutzt, so können unter Windows 2000 und Windows XP in der Datei *hosts* im Verzeichnis *<Systemroot>\system32\drivers\etc* die Zuordnungen von IP-Adressen zu frei wählbaren Hostnamen festgelegt werden.

```
java -cp build Lookup pc0823
IP-Nummer: 10.108.105.95
Hostname:  pc0823
```

Wird beim Aufruf des Programms eine IP-Adresse in Punkt-Notation anstelle eines Namens mitgegeben und konnte der Hostname nicht ermittelt werden, so wird als Hostname die vorgegebene IP-Adresse ausgegeben.

1.8 Aufgaben

1. Entwickeln Sie eine verbesserte Version von Programm 1.5 mit den folgenden Eigenschaften:

   - Sind einem Hostnamen mehrere IP-Adressen zugeordnet, so sollen alle angezeigt werden.
   - Wird statt eines Namens eine IP-Adresse beim Aufruf mitgegeben, so soll eine Fehlermeldung ausgegeben werden, wenn der Hostname nicht ermittelt werden konnte, ansonsten soll der zugeordnete Hostname angezeigt werden.

   Beispiel:

   ```
   java -cp build Lookup www.microsoft.com
   Hostname:  www.microsoft.com
   IP-Nummer: 207.46.244.188
   Hostname:  www.microsoft.com
   IP-Nummer: 207.46.245.92
   ...
   ```

2. Schreiben Sie ein Programm `ShowIP`, das die IP-Adresse und den Hostnamen des lokalen Rechners in einem Fenster anzeigt.

***Bild 1.17**: IP-Adresse und Hostname*

Dieses Programm kann z.B. genutzt werden, um die vom Provider dynamisch zugewiesene IP-Adresse bei einer Telefonverbindung anzuzeigen.

Um `ShowIP` mit einem Doppelklick auf einem grafischen Symbol starten zu können, muss nach der Compilierung eine JAR-Datei erzeugt werden:

```
jar cfm build/showip.jar manifest.txt -C build ShowIP.class
```

Die Datei *manifest.txt* hat den Inhalt:

```
Main-Class: ShowIP
```

Diese Textzeile muss mit einem Zeilenvorschub abgeschlossen sein.

Eine Verknüpfung mit *showip.jar* kann z.B. auf den Desktop gelegt werden.

2 Datenbankanwendungen mit JDBC

JDBC (Java Database Connectivity) ist die Standard-Schnittstelle für den Zugriff auf relationale Datenbanken mittels *SQL*. Der Kern von JDBC besteht aus einer Sammlung von Klassen und Interfaces, die im Paket `java.sql` zusammengefasst sind. Damit können Datenbankverbindungen aufgebaut, beliebige SQL-Anweisungen an die Datenbank geschickt und Abfrageergebnisse im Programm verarbeitet werden.

Java-Programme, die mittels JDBC auf eine Datenbank zugreifen, enthalten keinen datenbankspezifischen Code. Das ermöglicht den Austausch des Datenbanksystems (DBMS), ohne das Programm ändern zu müssen. JDBC stellt also eine *datenbankneutrale* Zugriffsschnittstelle bereit. JDBC ist eine *Abstraktionsschicht* zwischen Java-Programm und SQL.

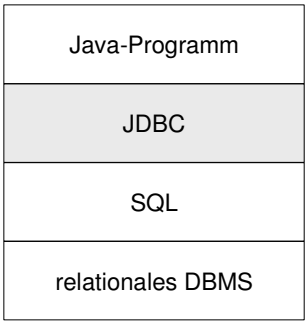


***Bild 2.1**: Schichtenmodell einer Datenbankanwendung*

Die zur Zeit der Drucklegung dieses Buches aktuelle Version JDBC 4.0 ist Teil von Java SE 6.0. In diesem Kapitel beschäftigen wir uns mit den Grundlagen von JDBC.

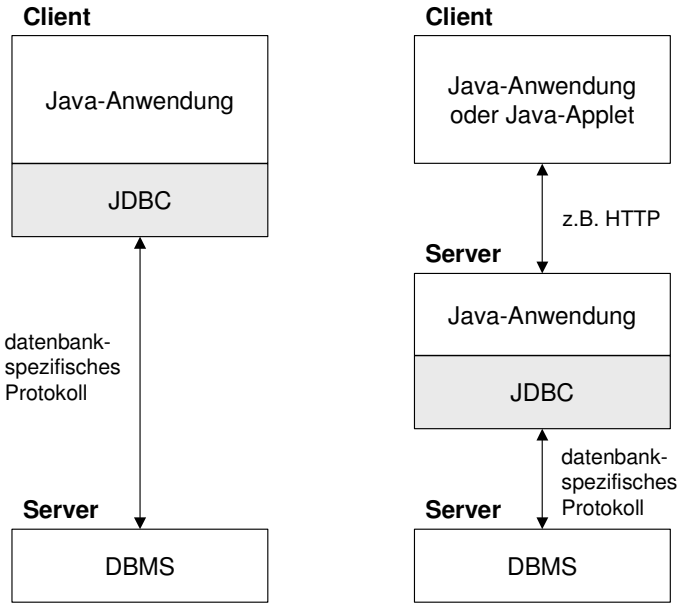
2.1 Die Architektur von JDBC-Anwendungen

Im einfachsten Fall wird JDBC im Client-Programm verwendet, das lokal oder über ein Netzwerk auf eine Datenbank zugreift (*zweistufige Architektur*).

Eine *dreistufige Architektur* trennt in der Regel die Anwendungslogik von der Benutzungsoberfläche und der Datenverwaltung. Ein Client (z.B. Webbrowser) kommuniziert über ein geeignetes Protokoll (z.B. HTTP) mit dem Applikationsserver,

der seinerseits auf den Datenbankserver zugreift. Bild 2.2 zeigt, wie verteilte Systeme mit JDBC realisiert werden können.

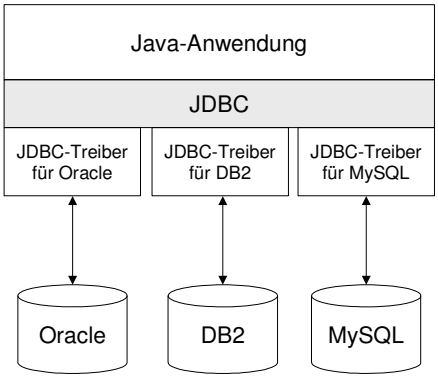
***Bild 2.2**: 2- und 3-stufige Architekturen*



Treiberkonzept

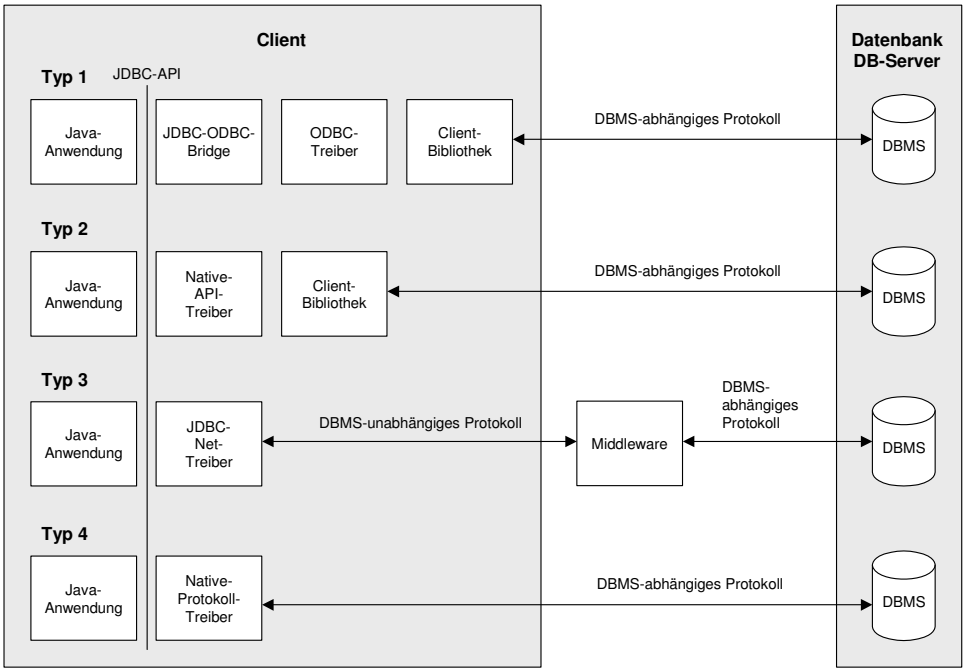
Jedes Datenbanksystem hat eine eigene, herstellerspezifische Zugriffsschnittstelle. Ein so genannter *JDBC-Treiber* übersetzt die JDBC-Methodenaufrufe in Aufrufe dieser Schnittstelle. So wird für jedes DBMS ein eigener Treiber benötigt. JDBC-Treiber existieren für die meisten relationalen DBMS. Beim JDBC-Treiber zu einem DBMS handelt es sich um eine Bibliothek von Klassen, die die von JDBC vorgegebenen Interfaces implementieren.

***Bild 2.3**: JDBC-Treiber*



Zur Realisierung von JDBC-Treibern gibt es grundsätzlich vier Möglichkeiten, die im Bild 2.4 zusammengefasst sind.

***Bild 2.4**: Typen von JDBC-Treibern*



JDBC-ODBC-Bridge und ODBC-Treiber *Typ1*

Treiber dieses Typs benutzen das ODBC-API (Open Database Connectivity) von Microsoft, um auf relationale Datenbanken zuzugreifen. Ein *JDBC-ODBC-Brückentreiber* wandelt alle JDBC-Aufrufe in ODBC-Aufrufe um. Somit lassen sich vorhandene ODBC-Treiber nutzen, um Datenbankanwendungen mit JDBC zu realisieren. Allerdings setzt das eine ODBC-Installation auf dem Client voraus. Die JDBC-ODBC-Bridge ist als Paket `sun.jdbc.odbc` Bestandteil von Java SE.

Native-API-Treiber *Typ2*

Ein solcher Treiber nutzt die DBMS-spezifische Programmierschnittstelle. Für jeden Client müssen clientseitige Bibliotheken (Binärcode) geladen werden.

JDBC-Net-Treiber *Typ3*

Treiber dieses Typs sind vollständig in Java realisiert. Sie nutzen eine zusätzliche Komponente (Middleware), die das DBMS-unabhängige Protokoll in ein DBMS-spezifisches Protokoll über-

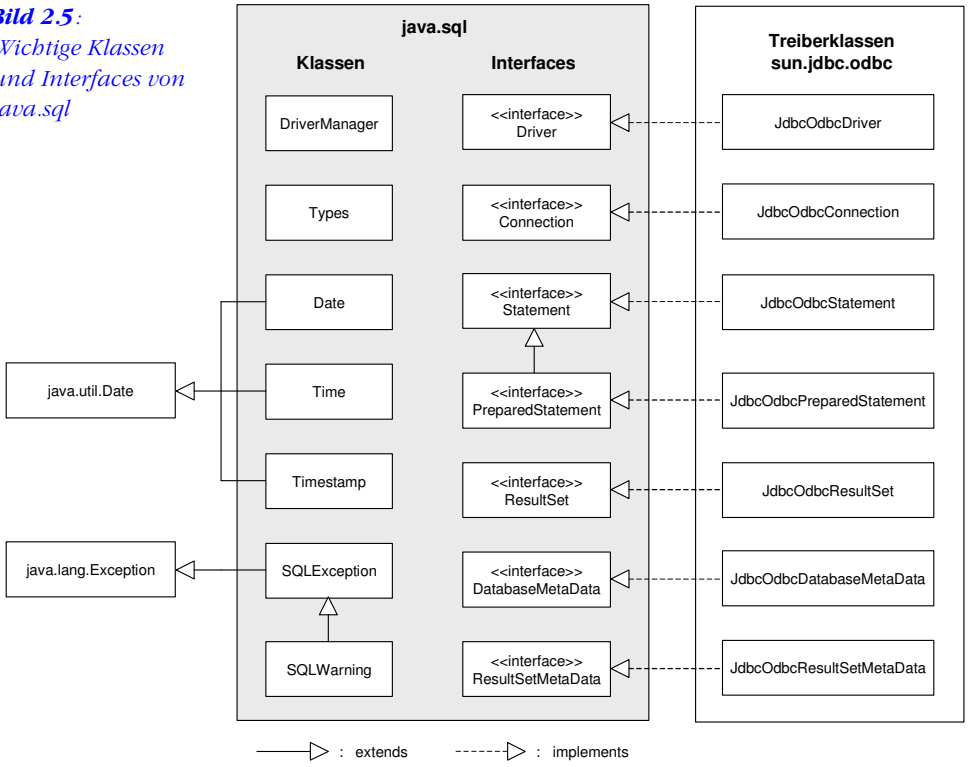
setzt. Diese Lösung ist sehr flexibel, da von einem Wechsel des DBMS nur die Middleware betroffen ist.

Typ 4

Native-Protokoll-Treiber
Diese vollständig in Java implementierten Treiber übersetzen JDBC-Aufrufe direkt in das DBMS-spezifische Protokoll. Sie werden meist von den DBMS-Herstellern selbst angeboten.

Bild 2.5 gibt einen Überblick über die wichtigsten Klassen und Interfaces des JDBC-API.

***Bild 2.5**: Wichtige Klassen und Interfaces von java.sql*



Die Klasse `DriverManager` handhabt das Laden des JDBC-Treibers und bietet Methoden zum Aufbau einer Datenbankverbindung.

Die wichtigsten Interfaces:

- `Connection` repräsentiert eine Datenbankverbindung.
- `Statement` wird verwendet, um SQL-Anweisungen über eine gegebene Datenbankverbindung auszuführen.
- `ResultSet` bietet Methoden, um auf das Ergebnis einer SQL-Abfrage zuzugreifen.

Bild 2.5 zeigt auch die Klassen der JDBC-ODBC-Bridge, die die Interfaces implementieren.

2.2 Erste Beispiele

2.2.1 Die Beispiel-Datenbank

Um verschiedene Aspekte von JDBC in den nächsten Abschnitten demonstrieren zu können, benutzen wir eine Datenbank, die Bücher verschiedener Verlage verwaltet. Bild 2.6 zeigt die Beziehung zwischen den Entitätstypen *verlag* und *buch*.

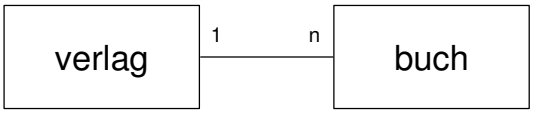


***Bild 2.6**: Datenmodell*

Ein Verlag hat verschiedene Bücher veröffentlicht. Ein Buch ist in genau einem Verlag erschienen.

In einer relationalen Datenbank werden die Entitätstypen als Tabellen aus Zeilen und Spalten implementiert. Die Spalten repräsentieren die verschiedenen Merkmale einer Entität, beim Buch also z.B. ISBN-Nummer, Autor, Titel usw. Jede Zeile enthält alle Angaben zu einer einzigen Entität.

isbn	autor	titel	ausgabe	seitenzahl	jahr	verlag_id	preis
3-15-001308-9	Dickens, Charles	Schwere Zeiten	Kartoniert	430	1989	9	7.6
3-15-001562-6	Dickens, Charles	Große Erwartungen	Kartoniert	751	1993	9	12.6
3-15-010606-0	Dickens, Charles	Der Weihnachtsabend	Gebunden	118	2006	9	6.9
3-257-20998-3	Dickens, Charles	Nikolas Nickleby	Kartoniert	728	1997	4	14.9
3-257-21034-5	Dickens, Charles	David Copperfield	Kartoniert	802	1982	4	14.9
3-257-21166-X	Dickens, Charles	Bleakhaus	Kartoniert	847	1984	4	14.9
3-257-21405-7	Dickens, Charles	Die Pickwickier	Kartoniert	650	2002	4	12.9
3-257-21406-5	Dickens, Charles	Martin Chuzzlewit	Kartoniert	813	1998	4	13.9
3-351-03044-4	Dickens, Charles	Weihnachten mit Dickens	Gebunden	160	2005	3	10
3-458-32655-3	Dickens, Charles	Harte Zeiten	Kartoniert	433	2004	6	11.5
3-458-32733-9	Dickens, Charles	Geschichte aus zwei Städten	Kartoniert	505	1987	6	12.5
3-458-32810-6	Dickens, Charles	Bleak House	Kartoniert	1030	1988	6	17
3-458-33004-6	Dickens, Charles	Nikolaus Nickleby	Kartoniert	1011	1991	6	14
3-491-96007-X	Dickens, Charles	Weihnachtserzählungen	Gebunden	591	2000	8	9.95
3-538-05349-9	Dickens, Charles	David Copperfield	Leinen	1023	1955	2	44.9
3-538-06656-6	Dickens, Charles	Die Pickwickier	Gebunden	1039	1997	2	12.9
3-538-06657-4	Dickens, Charles	Martin Chuzzlewit	Gebunden	1000	1997	2	12.9
3-538-06658-2	Dickens, Charles	Nicholas Nickleby	Gebunden	995	1997	2	12.9
3-538-06982-4	Dickens, Charles	Nicholas Nickleby	Gebunden	996	2004	2	24.9
3-7175-1976-X	Dickens, Charles	Das Geheimnis des Edwin Drood	Leinen	762	2001	7	24.9

***Bild 2.7**: Ausschnitt aus der Tabelle buch*

Primärschlüssel

Jede Tabelle enthält einen *Primärschlüssel*, der durch eine Spalte oder eine Kombination von mehreren Spalten repräsentiert wird. Der Wert des Primärschlüssels identifiziert höchstens eine Zeile der Tabelle. Die Schlüsselwerte einer Tabelle sind alle voneinander verschieden. Für die Tabelle *buch* bietet sich die Spalte *isbn* als Primärschlüssel an.

Fremdschlüssel

Um Verknüpfungen zwischen Tabellen herstellen zu können, werden so genannte Fremdschlüssel benötigt.

Ein *Fremdschlüssel* einer Tabelle wird durch eine Spalte oder durch eine Kombination von mehreren Spalten repräsentiert. Ein Fremdschlüssel ist in einer anderen Tabelle Primärschlüssel. Durch Vergleich von Fremdschlüsselwert und Primärschlüsselwert werden Beziehungen zwischen Zeilen der beiden Tabellen hergestellt.

So hat die Tabelle *buch* den Primärschlüssel *isbn*, die Tabelle *verlag* den Primärschlüssel *verlag_id*. Die Tabelle *buch* besitzt den Fremdschlüssel *verlag_id* (siehe Tabellenstruktur weiter unten).

Fremdschlüsselwerte innerhalb einer Tabelle sind in der Regel natürlich nicht eindeutig. Die 1:n-Beziehung zwischen *verlag* und *buch* sowie wird also durch die Fremdschlüssel-Primärschlüssel-Beziehung etabliert.

Im Weiteren setzen wir Grundkenntnisse der Datenbanksprache SQL voraus. Die SQL-Anweisungen zur Erstellung der drei Tabellen für das Datenbanksystem *MySQL* sind:

Tabelle verlag

```
create table verlag (
  verlag_id integer,
  verlag_name varchar(30),
  webadresse varchar(30),
  primary key (verlag_id)
)
```

Tabelle buch

```
create table buch (
  isbn varchar(17),
  autor varchar(30),
  titel varchar(80),
  ausgabe varchar(20),
  seitenzahl integer,
  jahr integer,
  verlag_id integer,
  preis double,
  bestand integer,
  stand datetime,
  primary key (isbn),
  foreign key (verlag_id) references verlag (verlag_id)
)
```

Die Programme dieses Kapitels wurden mit verschiedenen Datenbanksystemen getestet. Es wurden solche Systeme bevorzugt, die weit verbreitet bzw. ohne allzu großen Aufwand installiert werden können. Die folgende Tabelle führt die Datenbankprodukte und eingesetzten JDBC-Treiber mit dem Produktnamen und der Version auf. Bezugsquellen können am Ende des Buches nachgeschlagen werden.

Eingesetzte DB-Produkte und JDBC-Treiber

DBMS	JDBC-Treiber	Typ
Microsoft Access	JDBC-ODBC-Bridge 2.0001	1
MySQL Community Server 5.0.26	MySQL Connector/J 5.0.4	4
Apache Derby 10.2.1.6	Apache Derby Network Client JDBC Driver 10.2.1.6	4

Access, *MySQL* (mit Tabellentyp *InnoDB)* und *Apache Derby* unterstützen *Transaktionen* (siehe Kapitel 2.4.1) und stellen die *referentielle Integrität* sicher.

Java DB in JDK 6

Das Java SE 6 Development Kit (JDK) enthält *Apache Derby* unter der Bezeichnung *Java DB*.

2.2.2 Verbindungsaufbau

Alle Programme sind so geschrieben, dass die Datenbanksysteme leicht gewechselt werden können. Dazu sind die datenbankspezifischen Angaben in einer Konfigurationsdatei *dbconnect.properties* ausgelagert, die mit dem Texteditor bearbeitet werden kann.

In diesem Abschnitt zeigen wir den grundsätzlichen Aufbau einer JDBC-Anwendung. Wir setzen voraus, dass die Datenbank bereits eingerichtet ist und Testdaten enthält. In den folgenden Abschnitten dieses Kapitels werden dann der Aufbau und das Laden von Datenbanken mittels JDBC-Programmen ausführlich beschrieben. SQL-Skripte und Testdaten können der zum Download zur Verfügung gestellten Programmsammlung (Online-Service) entnommen werden.

Programm 2.1

Das erste Programmbeispiel gibt Informationen über die benutzte Datenbank und das eingesetzte DBMS aus. Datenbankspezifische Angaben (Treiber, Identifikation der Datenbank, User-Id und Passwort) befinden sich in der Datei *dbconnect.properties*:

dbconnect.properties

```
# MS Access
driver=sun.jdbc.odbc.JdbcOdbcDriver
url=jdbc:odbc:buecher
user=
password=

# MySQL
#driver=com.mysql.jdbc.Driver
#url=jdbc:mysql://localhost/buecher
#user=root
#password=root

# Apache Derby
#driver=org.apache.derby.jdbc.ClientDriver
#url=jdbc:derby://localhost/buecher
#user=root
#password=root
```

Hier sind die Angaben zur Access-Datenbank aktiviert. Das Zeichen # leitet einen Kommentar ein. Die Access-Datenbank muss mit dem Namen *buecher* als ODBC-Datenquelle registriert sein. Bei Windows muss zu diesem Zweck das Programm *Datenquellen (ODBC)* aus der Systemsteuerung aufgerufen und der MS-Access-Treiber sowie die Acess-Datenbank ausgewählt werden.

DBMetaData

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class DBMetaData {
  public static void main(String[] args) throws Exception {
    // DB-Parameter einlesen
    FileInputStream in = new FileInputStream("dbconnect.properties");
    Properties prop = new Properties();
    prop.load(in);
    in.close();

    String driver = prop.getProperty("driver");
    String url = prop.getProperty("url");
    String user = prop.getProperty("user");
    String password = prop.getProperty("password");

    // Treiber laden
    Class.forName(driver);

    // Verbindung zur DB herstellen
    Connection con = DriverManager.getConnection(url, user, password);

    DatabaseMetaData dbmd = con.getMetaData();
    System.out.println("URL: " + dbmd.getURL());
    System.out.println("UserName: " + dbmd.getUserName());
    System.out.println("DatabaseProductName: " +
      dbmd.getDatabaseProductName());
    System.out.println("DatabaseProductVersion: " +
```

```
      dbmd.getDatabaseProductVersion());
    System.out.println("DriverName: " + dbmd.getDriverName());
    System.out.println("DriverVersion: " + dbmd.getDriverVersion());

    con.close();
  }
}
```

Zunächst werden die Verbindungsparameter aus *dbconnect.properties* eingelesen und in entsprechenden Variablen abgelegt.

JDBC-Treiber laden

Die Treiberklasse (im Beispiel `sun.jdbc.odbc.JdbcOdbcDriver` für Access, `com.mysql.jdbc.Driver` für MySQL und `org.apache.derby.jdbc.ClientDriver` für Apache Derby) wird mit `Class.forName(...)` explizit geladen. Hierbei wird ein statischer Initialisierungsblock der Treiberklasse ausgeführt, der dafür sorgt, dass eine neue Instanz des Treibers beim Treibermanager (`java.sql.DriverManager`) registriert wird.

Sicherzustellen ist, dass die Klassen des JDBC-Treibers, zusammengefasst in einer Datei (z.B. `mysql-connector-java-5.0.4-bin.jar`) vom Class-Loader gefunden werden. Für unsere Beispiele wird der JDBC-Treiber in das Unterverzeichnis `jre\lib\ext` des Java-Installationsverzeichnisses kopiert ("installiertes optionales Paket"). Die Dokumentation zum jeweiligen Treiber gibt den Namen der zu ladenden Klasse an.

Verbindung zur Datenbank herstellen

Um eine Verbindung zur Datenbank aufbauen zu können, muss diese Datenbank genau identifiziert werden. Die Zieldatenbank wird in URL-Schreibweise angegeben:

```
jdbc:<subprotokoll>:<subname>
```

Hierbei bezeichnet das Subprotokoll die Art des verwendeten Treibers (z.B. `odbc`, `mysql` oder `derby`), Subname bezeichnet die eigentliche Datenbank. Bei Access ist das der Name der ODBC-Datenquelle, bei MySQL und Apache Derby ein URL (Uniform Resource Locator): `//localhost/buecher`. Die Datenbank liegt hier auf dem eigenen Rechner. Statt `localhost` kann auch der Name eines im Netz verbundenen anderen Rechners angegeben sein. Die Dokumentation zum Treiber enthält, was als Subprotokoll und Subname anzugeben ist.

Im Fall einer Access-Datenbank kann die Datenbankdatei auch direkt eingetragen werden, ohne sie im *ODBC-Datenquellen-Administrator* zu registrieren.

Beispiel (in einer Zeile zu erfassen):

```
jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};
DBQ=../db/Access/buecher.mdb
```

DriverManager

Die Klasse `java.sql.DriverManager` kann unterschiedliche Treiber zur selben Zeit handhaben. Wird eine Verbindung zu einer Datenbank angefordert, so versucht der Treibermanager, den passenden Treiber zu finden und zur Herstellung der Verbindung zu verwenden.

Die `DriverManager`-Methode `getConnection` stellt eine Verbindung zur Datenbank her.

```
static Connection getConnection(String url) throws SQLException

static Connection getConnection(String url, Properties info)
    throws SQLException

static Connection getConnection(
    String url, String user, String password) throws SQLException
```

Alle Methoden liefern ein `Connection`-Objekt und erwarten einen URL und ggf. weitere Angaben als Parameter. Datenbank-User und Passwort können angegeben werden. Bei der zweiten Methode sind die zusätzlichen Verbindungsparameter `user` und `password` im `Properties`-Objekt `info` eingetragen.

SQLException

Fehler werden in JDBC grundsätzlich als Ausnahmen der Klasse `java.sql.SQLException` ausgelöst.

Connection

Ein Objekt vom Schnittstellentyp `java.sql.Connection` repräsentiert eine Verbindung zur Datenbank.

Ein SQL-Anweisungsobjekt wird über die `Connection`-Schnittstelle erzeugt:

```
Statement createStatement() throws SQLException
```

Eine aktive Verbindung zur Datenbank wird mit der Methode `close` geschlossen:

```
void close() throws SQLException
```

Der Zustand einer Verbindung kann abgefragt werden:

```
boolean isClosed() throws SQLException
```

DatabaseMetaData

Das Interface `DatabaseMetaData` repräsentiert *Informationen über die Datenbank und das DBMS.*

Die folgende `Connection`-Methode liefert ein Metadaten-Objekt:

```
DatabaseMetaData getMetaData() throws SQLException
```

Die Schnittstelle umfasst weit über 100 Methoden. Hier werden nur einige aufgeführt, die auch im nächsten Beispiel verwendet werden.

```
String getURL() throws SQLException
```
liefert den URL der Datenbank.

`String` **`getUserName`**`() throws SQLException`
liefert den Namen des Datenbank-Users.

`String` **`getDatabaseProductName`**`() throws SQLException`
liefert den Produktnamen des Datenbanksystems.

`String` **`getDatabaseProductVersion`**`() throws SQLException`
liefert die Version des Datenbanksystems.

`String` **`getDriverName`**`() throws SQLException`
liefert den Namen des JDBC-Treibers.

`String` **`getDriverVersion`**`() throws SQLException`
liefert die Version des JDBC-Treibers.

Test

```
mkdir build
javac -sourcepath src -d build src/DBMetaData.java
java -cp build DBMetaData
```

Ausgabe:

```
URL: jdbc:odbc:buecher
UserName: admin
DatabaseProductName: ACCESS
DatabaseProductVersion: 04.00.0000
DriverName: JDBC-ODBC Bridge (ODBCJT32.DLL)
DriverVersion: 2.0001 (04.00.6019)
```

2.2.3 SELECT-Abfragen auswerten

Aufbau einer JDBC-Anwendung

Das Grundgerüst einer typischen JDBC-Anwendung besteht aus den folgenden Schritten:

1. JDBC-Treiber laden
2. Verbindung zur Datenbank herstellen
3. SQL-Anweisungsobjekt erzeugen
4. SQL-Anweisung ausführen
5. Ergebnisse der SQL-Anweisung verarbeiten
6. Ressourcen freigeben

Programm 2.2

Das zweite Programmbeispiel zeigt Informationen zu Büchern in Form einer sortierten Liste an. Die hierzu nötige SQL-Anweisung lautet:

```
select autor, titel, isbn from buch order by autor, titel
```

Buecherliste1

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class Buecherliste1 {
  public static void main(String[] args) throws Exception {
    // DB-Parameter einlesen
    FileInputStream in = new FileInputStream("dbconnect.properties");
    Properties prop = new Properties();
    prop.load(in);
    in.close();

    String driver = prop.getProperty("driver");
    String url = prop.getProperty("url");
    String user = prop.getProperty("user");
    String password = prop.getProperty("password");

    // Treiber laden
    Class.forName(driver);

    // Verbindung zur DB herstellen
    Connection con = DriverManager.getConnection(url, user, password);

    // SQL-Abfrage ausführen
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(
      "select autor, titel, isbn from buch order by autor, titel");

    // Ergebnisse ausgeben
    while (rs.next()) {
      System.out.println(rs.getString(1));
      System.out.println(rs.getString(2));
      System.out.println("ISBN " + rs.getString(3));
      System.out.println();
    }

    rs.close();
    stmt.close();
    con.close();
  }
}
```

Statement

Das Interface `java.sql.Statement` ist die Basisschnittstelle für alle SQL-Anweisungsformen.

Mit der `Statement`-Methode `executeQuery` wird eine SELECT-Anweisung ausgeführt:

```
ResultSet executeQuery(String sql) throws SQLException
```

Die Methode liefert ein `ResultSet`-Objekt mit dem Abfrageergebnis. Die SQL-Anweisung `sql` enthält kein Abschlusszeichen wie z.B. ";". Solche Abschlusszeichen können von DBMS zu DBMS variieren und werden demzufolge vom Treiber gesetzt.

`void close() throws SQLException`
gibt die mit der Ausführung gebundenen Ressourcen frei. Hierdurch wird auch das evtl. vorhandene `ResultSet`-Objekt geschlossen.

ResultSet

Das Ergebnis einer SQL-Abfrage (eine Relation mit Zeilen und Spalten) wird durch ein Objekt mit der Schnittstelle `ResultSet` repräsentiert.

Zum Navigieren über die Zeilen der Ergebnismenge wird die `ResultSet`-Methode `next` genutzt:

`boolean next() throws SQLException`

Ein interner Cursor zeigt auf die aktuelle Ergebniszeile. Zu Beginn ist der Cursor vor der ersten Zeile positioniert. `next` liefert so lange `true`, wie das Weitersetzen des Cursors erfolgreich war. Ist die Ergebnismenge leer oder das Tabellenende erreicht und keine weitere Zeile mehr vorhanden, so wird `false` zurückgeliefert.

`void close() throws SQLException`
gibt die genutzten Ressourcen frei.

Spaltenzugriff

Nachdem der interne Cursor auf eine Ergebniszeile positioniert wurde, kann auf die Spaltenwerte dieser Zeile *von links nach rechts* zugegriffen werden.

Für den Spaltenzugriff existieren verschiedene `ResultSet`-Methoden `getXxx`. Es kann über den Spaltenindex, der bei 1 beginnt, oder über den Spaltennamen zugegriffen werden. Zu beachten ist, dass der Spaltenindex sich auf die Spaltennummer in der Ergebnismenge und nicht in der Originaltabelle bezieht.

`getString` liefert den Spaltenwert als ein `String`-Objekt:

```
String getString(int columnIndex) throws SQLException
String getString(String columnName) throws SQLException
```

Programmausgabe:

```
Dickens, Charles
Bleak House
ISBN 3-458-32810-6

Dickens, Charles
Bleakhaus
ISBN 3-257-21166-X

Dickens, Charles
Das Geheimnis des Edwin Drood
ISBN 3-7175-1976-X
...
```

Programm 2.3

Im Unterschied zu Programm 2.2 wird im folgenden Programm auch der Verlag eines Buches ausgegeben:

```
Dickens, Charles
Bleak House
ISBN 3-458-32810-6 Insel
...
```

Die SQL-Anweisung enthält einen so genannten *Join* zwischen den Tabellen *buch* und *verlag*.

Buecherliste2 (Ausschnitt)

```
// SQL-Abfrage ausführen
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
  "select autor, titel, isbn, verlag_name " +
  "from buch inner join verlag " +
  "on buch.verlag_id = verlag.verlag_id " +
  "order by autor, titel");

// Ergebnisse ausgeben
while (rs.next()) {
  System.out.println(rs.getString(1));
  System.out.println(rs.getString(2));
  System.out.println("ISBN " + rs.getString(3) + " " +
     rs.getString(4));
  System.out.println();
}
```

Programm 2.4

Mit dem folgenden Programm kann gezielt nach einem Titel gesucht werden. Dazu muss als Kommandozeilenparameter der Titel oder nur ein Teil des Titels beim Aufruf angegeben werden.

Beispiel:

```
java -cp build Buecherliste3 Nickleby
```

Buecherliste3 (Ausschnitt)

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
  "select autor, titel, isbn, verlag_name " +
  "from buch inner join verlag " +
  "on buch.verlag_id = verlag.verlag_id " +
  "where titel like '%" + titel + "%' order by autor, titel");
```

2.3 JDBC-Datentypen

Bei den *SQL-Datentypen*, die von den verschiedenen Datenbanksystemen unterstützt werden, gibt es zum Teil erhebliche Unterschiede.

SQL arbeitet mit anderen Datentypen als Java. Beim Zugriff auf die Spaltenwerte und bei der Belegung von Parametern in JDBC-

Programmen (siehe `PreparedStatement` im nächsten Abschnitt) muss daher eine Abbildung zwischen beiden Typsystemen erfolgen.

JDBC-Datentypen

Um die am meisten verwendeten SQL-Datentypen einheitlich zu repräsentieren, wurden so genannte *JDBC-Datentypen* definiert, die in der Klasse `java.sql.Types` als Konstanten vom Typ `int` zusammengefasst sind:

```
public static final int XXX
```

Bild 2.8 gibt eine Übersicht über die JDBC-Datentypen und ihre Entsprechungen für die Datenbanksysteme Access, MySQL und Apache Derby.

Beim Lesen der Werte aus einem `ResultSet`-Objekt bzw. bei der Übergabe von Parametern an ein `PreparedStatement` (siehe nächsten Abschnitt) werden `getXxx`- bzw. `setXxx`-Methoden verwendet. Hier findet eine Konvertierung von SQL-Typen in Java-Typen bzw. umgekehrt statt. Bild 2.9 zeigt die Abbildung zwischen JDBC-Typen und Java-Typen.

***Bild 2.8**: Abbildung zwischen JDBC-Typen und SQL-Typen*

JDBC-Typ	SQL-Typ Access	SQL-Typ MySQL	SQL-Typ Derby
Integerzahlen			
TINYINT	BYTE	TINYINT	
SMALLINT	SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER, COUNTER	INTEGER, MEDIUMINT	INTEGER
BIGINT		BIGINT	BIGINT
Gleitkommazahlen			
REAL	REAL	FLOAT	REAL
FLOAT			
DOUBLE	DOUBLE, FLOAT	DOUBLE, REAL	DOUBLE, FLOAT
Festkommazahlen			
DECIMAL		DECIMAL(p,s), NUMERIC(p,s)	DECIMAL(p,s) NUMERIC(p,s)
NUMERIC	CURRENCY		

JDBC-Typ	SQL-Typ Access	SQL-Typ MySQL	SQL-Typ Derby
Datum und Uhrzeit			
DATE		DATE, YEAR	DATE
TIME		TIME	TIME
TIMESTAMP	DATE, TIME, DATETIME	TIMESTAMP, DATETIME	TIMESTAMP
Zeichenketten			
CHAR	CHAR(n)	CHAR(n)	CHAR(n)
VARCHAR	VARCHAR(n), TEXT	VARCHAR(n)	VARCHAR(n)
LONGVARCHAR	LONGCHAR, LONGTEXT	TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT	LONG VARCHAR
Binärdaten			
BIT	BIT	BOOLEAN	
BINARY	BINARY(n)		CHAR(n) FOR BIT DATA
VARBINARY	VARBINARY(n)	TINYBLOB	VARCHAR(n) FOR BIT DATA
LONGVARBINARY	LONGBINARY	BLOB, MEDIUMBLOB, LONGBLOB	LONG VARCHAR FOR BIT DATA

***Bild 2.9**: Abbildung zwischen JDBC- und Java-Datentypen*

JDBC-Typ	Java-Typ	Java-Objekttyp
TINYINT	`byte`	`Integer`
SMALLINT	`short`	`Integer`
INTEGER	`int`	`Integer`
BIGINT	`long`	`Long`
REAL	`float`	`Float`
FLOAT	`double`	`Double`
DOUBLE	`double`	`Double`

JDBC-Typ	Java-Typ	Java-Objekttyp
DECIMAL	java.math.BigDecimal	java.math.BigDecimal
NUMERIC	java.math.BigDecimal	java.math.BigDecimal
DATE	java.sql.Date	java.sql.Date
TIME	java.sql.Time	java.sql.Time
TIMESTAMP	java.sql.Timestamp	java.sql.Timestamp
CHAR	String	String
VARCHAR	String	String
LONGVARCHAR	String	String
BIT	boolean	Boolean
BINARY	byte[]	byte[]
VARBINARY	byte[]	byte[]
LONGVARBINARY	byte[]	byte[]

Die get-Methoden der Klasse `ResultSet` sind dann entsprechend:

getXxx

```
byte getByte(...)
short getShort(...)
int getInt(...)
long getLong(...)
float getFloat(...)
double getDouble(...)
java.math.BigDecimal getBigDecimal(...)
java.sql.Date getDate(...)
java.sql.Time getTime(...)
java.sql.Timestamp getTimestamp(...)
String getString(...)
boolean getBoolean(...)
byte[] getBytes(...)
```

Beim Aufruf einer get-Methode ist als Parameter eine Spaltennummer (Typ `int`) oder ein Spaltenname (Typ `String`) anzugeben.

Alle get-Methoden können die Ausnahme `SQLException` auslösen.

Die `ResultSet`-Methoden

```
Object getObject(int columnIndex) throws SQLException
Object getObject(String columnName) throws SQLException
```

liefern den Spaltenwert als ein Java-Objekt, dessen Typ dem entsprechenden JDBC-Datentyp aus der obigen Tabelle (Bild 2.9) entspricht. So wird z.B. für einen Spaltenwert vom JDBC-Typ `INTEGER` ein Objekt vom Typ `Integer` geliefert.

Datum/Zeit

Die Klassen `Date`, `Time` und `Timestamp` des Pakets `java.sql` sind Subklassen von `java.util.Date` und repräsentieren die JDBC-Datentypen `DATE`, `TIME` und `TIMESTAMP`. Objekte dieser Klassen enthalten Datumsangaben, Uhrzeitangaben bzw. Angaben zu Datum und Uhrzeit.

Date

`Date(long date)`
erzeugt ein `Date`-Objekt mittels des Zeitwerts `date` in Millisekunden (seit 01.01.1970, 00:00:00 GMT).

`void setTime(long date)`
setzt das Datum.

`static Date valueOf(String s)`
liefert ein `Date`-Objekt aus der Zeichenkette `s` im Format "`jjjj-mm-tt`".

`String toString()`
liefert eine Zeichenkette im Format "`jjjj-mm-tt`".

Time

`Time(long time)`
erzeugt ein `Time`-Objekt mittels des Zeitwerts `time` in Millisekunden (seit 01.01.1970, 00:00:00 GMT).

`void setTime(long time)`
setzt die Uhrzeit.

`static Time valueOf(String s)`
liefert ein `Time`-Objekt aus der Zeichenkette `s` im Format "`hh:mm:ss`".

`String toString()`
liefert eine Zeichenkette im Format "`hh:mm:ss`".

Timestamp

`Timestamp(long time)`
erzeugt ein `Timestamp`-Objekt mittels des Zeitwerts `time` in Millisekunden (seit 01.01.1970, 00:00:00 GMT).

`void setTime(long time)`
setzt Datum und Uhrzeit.

`long getTime()`
liefert Datum und Uhrzeit in Millisekunden (seit 01.01.1970, 00:00:00 GMT).

`static Timestamp valueOf(String s)`
liefert ein `Timestamp`-Objekt aus der Zeichenkette `s` im Format "`jjjj-mm-tt hh:mm:ss.f`", wobei `f` die Nanosekunden bezeichnet.

`String toString()`
liefert eine Zeichenkette im Format "`jjjj-mm-tt hh:mm: ss.f`".

`boolean after(Timestamp ts)`
prüft, ob diese Uhrzeit später als `ts` ist.

`boolean` **`before`**`(Timestamp ts)`
prüft, ob diese Uhrzeit früher als `ts` ist.

`int` **`compareTo`**`(Timestamp ts)`
vergleicht diese Uhrzeit mit `ts`.

`boolean` **`equals`**`(Timestamp ts)`
prüft, ob dieses `Timestamp`-Objekt inhaltlich gleich dem `Timestamp`-Objekt `ts` ist.

2.4 Ausführung von SQL-Anweisungen

In diesem Abschnitt beschäftigen wir uns mit der Änderung von Datenbankstrukturen und -inhalten, der Abfrage von Metadaten über Ergebnismengen zur Laufzeit und mit der Schnittstelle `PreparedStatement`.

2.4.1 Transaktionen

Im Fall von Änderungen steht der Begriff der *Transaktion* im Vordergrund.

Transaktion

Eine *Transaktion* ist eine Folge von Datenbankoperationen (SQL-Anweisungen), die die Datenbank von einem konsistenten Zustand in einen neuen konsistenten Zustand überführen. Die Anweisungen einer Transaktion werden entweder alle ausgeführt und abgeschlossen (*committed*) oder alle zurückgenommen (*rolled back*).

Transaktionen können automatisch oder manuell gesteuert werden. Eine JDBC-Anwendung ist standardmäßig im *Auto-Commit-Modus*: Jede einzelne SQL-Anweisung bildet eine Transaktion, die automatisch mit *Commit* bestätigt wird.

Um mehrere Anweisungen innerhalb einer einzigen Transaktion ausführen zu können, muss *Auto-Commit* ausgeschaltet werden.

Das Interface `Connection` deklariert Methoden zur Steuerung von Transaktionen.

Auto-Commit

`void` **`setAutoCommit`**`(boolean autoCommit) throws SQLException`
schaltet *Auto-Commit* ein (`true`) oder aus (`false`).

`boolean` **`getAutoCommit`**`() throws SQLException`
prüft, ob *Auto-Commit* eingeschaltet ist.

Commit und Rollback

`void` **`commit`**`() throws SQLException`
zeigt an, dass die Transaktion abgeschlossen werden soll und alle Änderungen in der Datenbank permanent gespeichert werden sollen.

```
void rollback() throws SQLException
```
bricht die aktive Transaktion ab, alle bisherigen Änderungen innerhalb dieser Transaktion werden rückgängig gemacht.

Transaktionen werden automatisch begonnen. Die erste SQL-Anweisung nach einem *Commit* oder *Rollback* gehört schon zu einer neuen Transaktion. Eine manuell gesteuerte, noch nicht abgeschlossene Transaktion wird explizit mit der Methode `commit` oder `rollback` geschlossen.

2.4.2 Datenbankänderungen

SQL-Anweisungen, die die Datenbankstruktur betreffen (z.B. CREATE TABLE) und die Änderungsbefehle INSERT, UPDATE und DELETE werden mit Hilfe der folgenden `Statement`-Methode an die Datenbank geschickt:

```
int executeUpdate(String sql) throws SQLException
```
Im Fall von INSERT, UPDATE oder DELETE liefert die Methode die Anzahl der von der Änderung betroffenen Zeilen.

Programm 2.5

Programm 2.5 richtet die im Kapitel 2.2 eingeführten Tabellen *verlag* und *buch* ein. Die *Properties*-Datei *tables.properties* enthält die Verweise auf die entsprechenden SQL-Skripte.

tables.properties

```
tables ../db/Access/create_verlag.sql \
       ../db/Access/create_buch.sql

#tables ../db/MySQL/create_verlag.sql \
#       ../db/MySQL/create_buch.sql

#tables ../db/Derby/create_verlag.sql \
#       ../db/Derby/create_buch.sql
```

Das Programm ist unabhängig von einem konkreten Datenmodell und somit universell einsetzbar.

CreateTables

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class CreateTables {
  public static void main(String[] args) {
    Connection con = null;
    try {
      FileInputStream in = new FileInputStream(
         "dbconnect.properties");
      Properties dbProp = new Properties();
      dbProp.load(in);
      in.close();
```

```
      String driver = dbProp.getProperty("driver");
      String url = dbProp.getProperty("url");
      String user = dbProp.getProperty("user");
      String password = dbProp.getProperty("password");

      Class.forName(driver);
      con = DriverManager.getConnection(url, user, password);

      in = new FileInputStream("tables.properties");
      Properties tablesProp = new Properties();
      tablesProp.load(in);
      in.close();

      StringTokenizer tables = new StringTokenizer(
         tablesProp.getProperty("tables"));

      while (tables.hasMoreTokens()) {
        BufferedReader reader = new BufferedReader(
          new FileReader(tables.nextToken()));
        String line;
        StringBuilder sb = new StringBuilder();
        while ((line = reader.readLine()) != null) {
          sb.append(line);
        }
        reader.close();

        Statement stmt = con.createStatement();
        stmt.executeUpdate(sb.toString());
        stmt.close();
      }
    }
    catch (Exception e) {
      System.err.println(e);
    }
    finally {
      try {
        con.close();
      }
      catch (SQLException e) {
        System.err.println(e);
      }
    }
  }
}
```

In vielen Datenbanksystemen (wie auch bei MySQL) wird eine Transaktion durch *create table* automatisch abgeschlossen. Deshalb bleibt hier *Auto-Commit* eingeschaltet.

Programm 2.6

Das folgende Programm ermöglicht das Laden von Tabellen aus externen Quellen.

Importdatei

Die Importdatei enthält pro Zeile einen in die Tabelle einzutragenden Datensatz, wobei die Spaltenwerte in der gleichen Reihenfolge auftreten, in der die Tabellenspalten mit der SQL-Anweisung "CREATE TABLE" definiert wurden. Die Werte in der Datei sind durch ein Sonderzeichen (z.B. `;` oder das Tabulatorzeichen), das in keinem der Werte enthalten sein darf, getrennt (*Feldtrenner*).

Beispiel:
Die Importdatei für die Tabelle *verlag* hat den Inhalt:

```
1;Anaconda;\N
2;Artemis & Winkler;www.patmos.de
3;Aufbau;www.aufbau-verlag.de
4;Diogenes;www.diogenes.ch
...
```

Feldtrenner ist hier `;`. Evtl. nicht vorhandene Werte sind durch `\N` zu kennzeichnen. Diese entsprechen den NULL-Werten in der Datenbank. Jede Zeile enthält drei Werte, die den Spalten *verlag_id*, *verlag_name* und *webadresse* entsprechen.

Es folgen einige Konventionen für die Notierung der Werte in der Importdatei:

- Zeichenketten werden ohne Begrenzer wie z.B. `"` usw. eingegeben.
- Numerische Werte werden wie Literale in Java notiert.
- Nicht vorhandene Werte (NULL-Werte) werden durch `\N` gekennzeichnet.
- Formate für Datum, Uhrzeit und Zeitstempel sind "`jjjj-mm-tt`", "`hh:mm:ss`" bzw. "`jjjj-mm-tt hh:mm:ss`".

Das Programm ist allgemeingültig geschrieben und kann zum Laden beliebiger Tabellen genutzt werden. Es wird mit drei Parametern aufgerufen:

1. Name des Verzeichnisses, in dem sich die Importdateien befinden ("." kennzeichnet das aktuelle Verzeichnis)
2. Feldtrenner, z.B. `;` oder `\t` (falls das Tabulatorzeichen benutzt wurde)
3. Name der Tabelle

Der Name der Importdatei für die Tabelle *xxx* muss dann *xxx.txt* lauten.

Das Programm generiert für jede eingelesene Zeile der Importdatei eine INSERT-Anweisung. Da die Tabellenstruktur (insbesondere die verwendeten SQL-Datentypen) dem Programm zur Übersetzungszeit nicht bekannt ist, müssen Informationen hierüber zur Laufzeit abgefragt werden.

Hierzu wird die `ResultSet`-Methode `getMetaData` verwendet, die ein `ResultSetMetaData`-Objekt bereitstellt, das die gewünschten Informationen enthält:

```
ResultSetMetaData getMetaData() throws SQLException
```

ResultSetMetaData

Das Interface `ResultSetMetaData` deklariert u.a. Methoden, um den JDBC-Datentyp einer Spalte, die Anzahl der Spalten und einen Spaltennamen der Ergebnisrelation abzufragen.

```
int getColumnType(int column) throws SQLException
int getColumnCount() throws SQLException
String getColumnLabel(int column) throws SQLException
```

Import

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class Import {
  public static void main(String[] args) {
    String dir = args[0];
    String delimiter = args[1];
    String table = args[2];

    if (delimiter.equals("\\t"))
      delimiter = "\t";

    BufferedReader tab = null;
    Connection con = null;
    try {
      tab = new BufferedReader(
        new FileReader(dir + "/" + table + ".txt"));

      FileInputStream in = new FileInputStream(
         "dbconnect.properties");
      Properties prop = new Properties();
      prop.load(in);
      in.close();

      String driver = prop.getProperty("driver");
      String url = prop.getProperty("url");
      String user = prop.getProperty("user");
      String password = prop.getProperty("password");

      Class.forName(driver);
      con = DriverManager.getConnection(url, user, password);
      con.setAutoCommit(false);
      Statement stmt = con.createStatement();
```

```
      // Ermittlung der Metadaten zur Feststellung der Spaltentypen
      ResultSet rs = stmt.executeQuery(
        "select * from " + table + " where 0 = 1");
      ResultSetMetaData rsmd = rs.getMetaData();

      StringTokenizer st;
      String line;
      while ((line = tab.readLine()) != null) {
        if (line.length() == 0)
          continue;

        st = new StringTokenizer(line, delimiter);
        StringBuilder sql = new StringBuilder(
          "insert into " + table + " values (");
        String s;
        int i = 0;
        while (st.hasMoreTokens()) {
          i++;
          s = st.nextToken();
          sql.append(getValue(rsmd.getColumnType(i), s));
          if (st.countTokens() != 0)
            sql.append(",");
        }
        sql.append(")");

        stmt.executeUpdate(sql.toString());
      }

      stmt.close();
      con.commit();
      System.out.println("Daten wurden in Tabelle '" + table +
        "' importiert");
    }
    catch (Exception e) {
      System.err.println(e);
      try {
        if (con != null)
          con.rollback();
      }
      catch (SQLException ex) {
        System.err.println(ex);
      }
    }
    finally {
      try {
        if (con != null)
          con.close();
        if (tab != null)
          tab.close();
      }
      catch (Exception e) {
        System.err.println(e);
      }
    }
  }
```

```
  // typgerechte Aufbereitung der Spaltenwerte für INSERT
  private static String getValue(int type, String s) {
    // aus \N wird null
    if (s.equals("\\N"))
      return "null";
    else if (type == Types.CHAR || type == Types.VARCHAR ||
      type == Types.LONGVARCHAR) {
      // der einfache Anführungsstrich ' wird verdoppelt
      s = s.replaceAll("'", "''");
      return "'" + s + "'";
    }
    else if (type == Types.DATE)
      return "{d '" + s + "'}";
    else if (type == Types.TIME)
      return "{t '" + s + "'}";
    else if (type == Types.TIMESTAMP) {
      // bei Access entsprechen die SQL-Typen DATE, TIME, DATETIME
      // alle dem JDBC-Typ TIMESTAMP
      if (s.indexOf(' ') != -1)
        return "{ts '" + s + "'}";
      else if (s.indexOf('-') != -1)
        return "{d '" + s + "'}";
      else
        return "{t '" + s + "'}";
    }
    else
      return s;
  }
}
```

Um Informationen über die JDBC-Datentypen der Tabellenspalten zu erhalten, wird zunächst eine SQL-Abfrage der Form

```
select * from tabelle where 0 = 1
```

ausgeführt, die eine leere Ergebnismenge liefert.

Die `ResultSet`-Methode `getMetaData` liefert dann ein `ResultSetMetaData`-Objekt.

Mit Hilfe von `StringTokenizer` wird nun jede Zeile in ihre Werte zerlegt und die INSERT-Anweisung aufgebaut:

```
insert into tabelle values (wert1, ...)
```

Die Hilfsmethode `getValue(int type, String s)` sorgt für die SQL-konforme Aufbereitung des eingelesenen Wertes. Hier liegen die folgenden Regeln zugrunde:

Ist `s` gleich `"\N"`, so liefert die Methode `"null"` zurück.

Ist der JDBC-Typ `type` gleich `CHAR`, `VARCHAR` oder `LONGVARCHAR`, wird der Wert in einfache Hochkommata eingeschlossen. Ein einfaches Hochkomma `'` im Wert selbst wird verdoppelt.

Ist `type` gleich `DATE`, `TIME` oder `TIMESTAMP`, so werden *Escape-Klauseln* der Form `{d 'xxx'}`, `{t 'xxx'}` bzw. `{ts 'xxx'}` zurück-

geliefert, wobei `xxx` für das Literal für Datum, Zeit bzw. Zeitstempel steht (siehe obige Beschreibung der Importdatei).

Test

Wegen der bestehenden Fremdschlüssel-Primärschlüssel-Beziehung müssen die Tabellen in der folgenden Reihenfolge geladen werden:

```
java -cp build Import . ; verlag
java -cp build Import . ; buch
```

Batch-Updates

Mehrere INSERT- oder UPDATE-Anweisungen können in einem `Statement`-Objekt mit `addBatch` gesammelt werden und dann in einem Zug mit `executeBatch` ausgeführt werden. Diese Vorgehensweise ist in der Regel effizienter als die einzelne Ausführung von SQL-Anweisungen.

Hier die beiden `Statement`-Methoden:

```
void addBatch(String sql) throws SQLException
int[] executeBatch() throws SQLException
```

Das `int`-Array enthält für jede SQL-Anweisung die Angabe darüber, wie viele Datensätze eingefügt bzw. verändert wurden, den Wert `Statement.SUCCESS_NO_INFO` oder `Statement.EXECUTE_FAILED` im Fehlerfall.

Programm 2.7

Das nächste Programm exportiert die Daten einer Tabelle im oben beschriebenen Import-Format. Damit können die Daten der Datenbank in einem Textformat gesichert werden.

Export

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class Export {
  public static void main(String[] args) {
    String dir = args[0];
    String delimiter = args[1];
    String table = args[2];

    if (delimiter.equals("\\t"))
      delimiter = "\t";

    PrintWriter out = null;
    Connection con = null;
    try {
      out = new PrintWriter(new FileWriter(
        dir + "/" + table + ".txt"), true);

      FileInputStream in = new FileInputStream(
         "dbconnect.properties");
      Properties prop = new Properties();
```

```
      prop.load(in);
      in.close();

      String driver = prop.getProperty("driver");
      String url = prop.getProperty("url");
      String user = prop.getProperty("user");
      String password = prop.getProperty("password");

      Class.forName(driver);
      con = DriverManager.getConnection(url, user, password);
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery("select * from " + table);
      ResultSetMetaData rsmd = rs.getMetaData();
      int n = rsmd.getColumnCount();

      while (rs.next()) {
        for (int i = 1; i <= n; i++) {
          String value = rs.getString(i);
          if (rs.wasNull())
            out.print("\\N");
          else {
            out.print(value);
          }
          if (i < n)
            out.print(delimiter);
        }
        out.println();
      }

      System.out.println("Tabelle '" + table + "' wurde exportiert");
      stmt.close();
    }
    catch (Exception e) {
      System.err.println(e);
    }
    finally {
      try {
        if (con != null)
          con.close();
        if (out != null)
          out.close();
      }
      catch (Exception e) {
        System.err.println(e);
      }
    }
  }
}
```

Nullwert abfragen

Um feststellen zu können, ob eine Spalte der Ergebnisrelation keinen Eintrag hat (SQL-Wert `NULL`), muss zunächst mit einer `getXxx`-Methode auf die Spalte zugegriffen werden. Dann kann mit der folgenden `ResultSet`-Methode abgefragt werden, ob der letzte gelesene Wert ein *Nullwert* war:

```
boolean wasNull() throws SQLException
```

Test

```
java -cp build Export . ; buch
```

Die Daten werden in der Datei *buch.txt* im aktuellen Verzeichnis gespeichert.

2.4.3 Prepared Statements

Wird ein und dieselbe SQL-Anweisung mehrfach (mit verschiedenen Parameterwerten) ausgeführt, so können Laufzeitvorteile dadurch erzielt werden, dass statt eines `Statement`-Objekts ein Objekt vom Typ des Subinterfaces `PreparedStatement` verwendet wird.

Ein `PreparedStatement`-Objekt wird durch Aufruf der `Connection`-Methode `prepareStatement` erzeugt:

```
PreparedStatement prepareStatement(String sql) throws SQLException
```

Der String `sql` wird zum DBMS geschickt, dort compiliert und für die Ausführung vorbereitet.

Evtl. Parameterwerte werden in diesem String nicht angegeben, sondern nur durch den Platzhalter `?` gekennzeichnet. Somit wird der String vom DBMS nur einmal geparst und auf Korrektheit geprüft. Bei jeder späteren Ausführung werden nur noch die Parameterwerte gefüllt.

setXxx

Analog zu den `getXxx`-Methoden von `ResultSet` (siehe Kapitel 2.3) stellt `PreparedStatement` `setXxx`-Methoden bereit, mit denen die Parameterwerte gesetzt werden können.

```
void setByte(int idx, byte x)
void setShort(int idx, short x)
void setInt(int idx, int x)
void setLong(int idx, long x)
void setFloat(int idx, float x)
void setDouble(int idx, double x)
void setBigDecimal(int idx, BigDecimal x)
void setDate(int idx, Date x)
void setTime(int idx, Time x)
void setTimestamp(int idx, Timestamp x)
void setString(int idx, String x)
void setBoolean(int idx, boolean x)
void setBytes(int idx, byte[] x)
```

`idx` kennzeichnet den zu ersetzenden Parameter mit Hilfe seiner Positionsnummer, von links nach rechts, beginnend mit 1. Alle set-Methoden können eine `SQLException` auslösen.

```
void setObject(int idx, Object value) throws SQLException
```

setzt den Wert des Parameters an der Stelle `idx` mit Hilfe des Objekts `obj`. Dabei wird `value` in einen Wert des entsprechenden SQL-Typs konvertiert (siehe Bild 2.8 und 2.9).

Nullwerte können mit `setNull` übergeben werden, wobei eine passende Konstante aus `java.sql.Types` anzugeben ist:

```
void setNull(int parameterIndex, int sqlType) throws SQLException
```

Das Interface `PreparedStatement` enthält auch die entsprechenden Methoden zur Ausführung der SQL-Anweisung:

```
ResultSet executeQuery() throws SQLException
int executeUpdate() throws SQLException
```

Batch-Updates

Batch-Updates können auch mit `PreparedStatement` ausgeführt werden. Mit Hilfe der `PreparedStatement`-Methode

```
void addBatch() throws SQLException
```

werden Parametersätze, die mit den `setXxx`-Methoden angelegt wurden, eingetragen.

Die einzelnen Operationen werden dann wieder am Schluss in einem Zug mit Hilfe der Methode `executeBatch` (siehe oben) ausgeführt.

Programm 2.8

Das folgende Programm nutzt die eben behandelten Möglichkeiten, um den Lagerbestand zu erhöhen bzw. zu vermindern. Die folgende UPDATE-Anweisung wird verwendet:

```
update buch set bestand = bestand + ?, stand = ? where isbn = ?
```

ISBN-Nummer und Mengenwert werden aus der Eingabedatei *bestand.txt* (Aufrufparameter) gelesen.

Bild 2.10 zeigt den Datenbankzustand vor der Änderung.

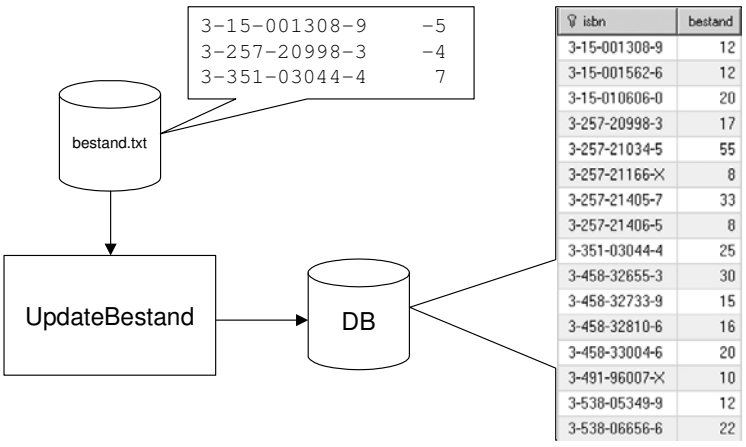


***Bild 2.10**: Update des Lagerbestands*

UpdateBestand

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class UpdateBestand {
  public static void main(String[] args) {
    String datei = args[0];

    BufferedReader input = null;
    Connection con = null;
    try {
      input = new BufferedReader(new FileReader(datei));

      FileInputStream in = new FileInputStream(
         "dbconnect.properties");
      Properties prop = new Properties();
      prop.load(in);
      in.close();

      String driver = prop.getProperty("driver");
      String url = prop.getProperty("url");
      String user = prop.getProperty("user");
      String password = prop.getProperty("password");

      Class.forName(driver);
      con = DriverManager.getConnection(url, user, password);
      con.setAutoCommit(false);

      PreparedStatement stmt = con.prepareStatement(
        "update buch set bestand = bestand + ?, " +
        "stand = ? where isbn = ?");

      String isbn;
      int bestand;
      int count;
      StringTokenizer st;
      String line;

      while ((line = input.readLine()) != null) {
        st = new StringTokenizer(line);
        isbn = st.nextToken();
        bestand = Integer.parseInt(st.nextToken());
        stmt.setInt(1, bestand);
        stmt.setTimestamp(2, new Timestamp(
          System.currentTimeMillis()));
        stmt.setString(3, isbn);
        count = stmt.executeUpdate();

        if (count > 0)
          System.out.println(isbn + ": Bestand aktualisiert");
      }

      con.commit();
      stmt.close();
    }
    catch (Exception e) {
      System.err.println(e);
```

```
      try {
        if (con != null)
          con.rollback();
      }
      catch (SQLException ex) {
        System.err.println(ex);
      }
    }
    finally {
      try {
        if (con != null)
          con.close();
        if (input != null)
          input.close();
      }
      catch (Exception e) {
        System.err.println(e);
      }
    }
  }
}
```

2.5 Ein einfaches Frontend für SQL-Datenbanken

In Kapitel 2.2 wurden bereits Metadaten über die Datenbank (`DatabaseMetaData`) genutzt. Hier folgen zwei weitere Methoden:

```
ResultSet getTables(String catalog, String schemaPattern,
    String tableNamePattern, String[] types) throws SQLException
```

liefert eine Beschreibung der Tabellen der Datenbank. Im Beispiel rufen wir die Methode mit den Parametern `null`, `null`, `"%"`, `null` auf. Das `ResultSet`-Objekt enthält u.a. die Spalten `TABLE_NAME` und `TABLE_TYPE`.

```
ResultSet getColumns(String catalog, String schemaPattern,
    String tableNamePattern, String columnNamePattern)
    throws SQLException
```

liefert eine Beschreibung der Tabellenspalten. Im Beispiel rufen wir die Methode mit den Parametern `null`, `null`, `tabelle`, `"%"` auf. Das `ResultSet`-Objekt enthält u.a. die Spalten `COLUMN_NAME`, `DATA_TYPE` (JDBC-Datentyp aus `java.sql.Types`), `TYPE_NAME`, `COLUMN_SIZE`.

Programm 2.9

Das folgende Programm bietet ein *Frontend* für SQL-Datenbanken, mit dem beliebige SQL-Anweisungen (SELECT, INSERT, UPDATE, DELETE) an eine Datenbank geschickt werden können.

Die Abfrageergebnisse können auf Wunsch in der Datei *log.txt* protokolliert werden. Das Programm bietet neben den SQL-Anweisungen noch die folgenden Eingabemöglichkeiten:

`show tables`	alle Tabellennamen anzeigen
`show table xxx`	Metadaten zur Tabelle `xxx` anzeigen
`log on`	Protokollierung einschalten
`log off`	Protokollierung ausschalten
`q`	Programm beenden

Unbekannte Anweisungen ausführen

In diesem Programm nutzen wir die `Statement`-Methode `execute`, um beliebige, zur Compilierungszeit unbekannte SQL-Anweisungen an die Datenbank zu schicken:

```
boolean execute(String sql) throws SQLException
```

Der Rückgabewert ist `true`, wenn ein Abfrageergebnis vorliegt, und `false`, wenn es sich um eine Änderung handelt.

Mit den beiden `Statement`-Methoden `getResultSet` und `getUpdateCount` kann die Ergebnismenge bzw. die Anzahl der geänderten Zeilen ermittelt werden:

```
ResultSet getResultSet() throws SQLException
int getUpdateCount() throws SQLException
```

Auch die Schnittstelle `PreparedStatement` enthält eine `execute`-Methode:

```
boolean execute() throws SQLException
```

SQLException

Bei schwerwiegenden Fehlern melden die meisten JDBC-Methoden einen Fehler durch eine Ausnahme der Klasse `java.sql.SQLException` (Subklasse von `Exception`). Dabei können mehrere Fehlermeldungen in einem Objekt vom Typ `SQLException` verpackt sein.

`SQLException` enthält folgende Methoden:

```
String getSQLState()
```
liefert eine Fehlerbeschreibung nach X/OPEN- bzw. SQL99-Konventionen.

```
int getErrorCode()
```
liefert einen herstellerspezifischen Fehlercode.

```
SQLException getNextException()
```
liefert das nächste Ausnahme-Objekt, falls mehrere Fehlermeldungen vorliegen.

Sql

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class Sql {
  private static Connection con;
  private static boolean log;
  private static BufferedReader input;
  private static PrintWriter output;

  public static void main(String[] args) {
    try {
      FileInputStream in = new FileInputStream(
         "dbconnect.properties");
      Properties prop = new Properties();
      prop.load(in);
      in.close();

      String driver = prop.getProperty("driver");
      String url = prop.getProperty("url");
      String user = prop.getProperty("user");
      String password = prop.getProperty("password");

      Class.forName(driver);
      con = DriverManager.getConnection(url, user, password);

      input = new BufferedReader(new InputStreamReader(System.in));
      String query;

      while (true) {
        System.out.print("> ");
        query = input.readLine();

        if (query.equals("q"))
          break;

        if (query.equals("log on")) {
          if (output == null) {
            output = new PrintWriter(
              new FileWriter("log.txt"), true);
          }
          log = true;
          continue;
        }

        if (query.equals("log off")) {
          log = false;
          continue;
        }

        if (log)
          output.println("Query: " + query);

        try {
          process(query);
        }
```

```
      catch (SQLException e) {
        printSQLException(e);
      }
    }
  }
  catch (Exception e) {
    System.err.println(e);
  }
  finally {
    try {
      if (con != null)
        con.close();
      if (input != null)
        input.close();
      if (output != null)
        output.close();
    }
    catch (SQLException e) {
      printSQLException(e);
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
}

private static void printSQLException(SQLException e) {
  System.err.println("SQLException:");
  while (e != null) {
    System.err.println("SQLState: " + e.getSQLState());
    System.err.println("Message: " + e.getMessage());
    System.err.println("ErrorCode: " + e.getErrorCode());
    e = e.getNextException();
  }
}

private static void process(String query) throws SQLException {
  if (query.equals("show tables")) {
    showTables();
    return;
  }

  if (query.startsWith("show table ")) {
    String table = query.substring(11).trim();
    showTable(table);
    return;
  }

  Statement stmt = con.createStatement();

  boolean isResultSet = stmt.execute(query);
  if (isResultSet) {
    ResultSet rs = stmt.getResultSet();
    ResultSetMetaData rsmd = rs.getMetaData();
    int numCols = rsmd.getColumnCount();
    int count = 0;
```

```
      while (rs.next()) {
        count++;
        System.out.println("# " + count);
        if (log)
          output.println("# " + count);

        for (int i = 1; i <= numCols; i++) {
          String line = rsmd.getColumnLabel(i) + ": " +
            rs.getString(i);
          System.out.println(line);
          if (log)
            output.println(line);
        }
      }

      if (log)
        output.println();

      rs.close();
    }
    else {
      int updateCount = stmt.getUpdateCount();
      System.out.println("UpdateCount: " + updateCount);
      if (log) {
        output.println("UpdateCount: " + updateCount);
        output.println();
      }
    }

    stmt.close();
  }

  private static void showTables() throws SQLException {
    DatabaseMetaData dbmd = con.getMetaData();
    ResultSet rs = dbmd.getTables(null, null, "%", null);
    String table, type;

    while (rs.next()) {
      table = rs.getString("TABLE_NAME");
      type = rs.getString("TABLE_TYPE");
      System.out.println(type + ": " + table);
      if (log)
        output.println(table);
    }

    if (log)
      output.println();

    rs.close();
  }

  private static void showTable(String table) throws SQLException {
    DatabaseMetaData dbmd = con.getMetaData();
    ResultSet rs = dbmd.getColumns(null, null, table, "%");
    String name, type, size;
```

```
    while (rs.next()) {
      name = rs.getString("COLUMN_NAME");
      type = rs.getString("TYPE_NAME");
      size = rs.getString("COLUMN_SIZE");
      System.out.println(name + " " + type + "(" + size + ")");
      if (log)
        output.println(name + " " + type + "(" + size + ")");
    }

    if (log)
      output.println();

    rs.close();
  }
}
```

Test mit MySQL-Datenbank

```
java -cp build Sql

> log on
> show tables
TABLE: buch
TABLE: verlag
> show table buch
isbn varchar(17)
autor varchar(30)
titel varchar(80)
ausgabe varchar(20)
seitenzahl int(11)
jahr int(11)
verlag_id int(11)
preis double(22)
bestand int(11)
stand datetime(19)
> select * from buch where titel like '%Twist%'
# 1
isbn: 3-7466-2200-X
autor: Dickens, Charles
titel: Oliver Twist
ausgabe: Kartoniert
seitenzahl: 564
jahr: 2005
verlag_id: 3
preis: 9.95
bestand: 50
stand: 2006-08-20 00:00:00.0
> log off
> q
```

2.6 Speicherung großer Objekte

Wir wollen nun die Struktur der Tabelle *buch* um eine zusätzliche Spalte, die ein Bild des Covers aufnehmen kann, ergänzen.

Die hierfür geeignete SQL-Anweisung lauten:

Access: `alter table buch add bild longbinary`

MySQL: `alter table buch add bild longblob`

Apache Derby: `alter table buch add bild blob`

Große Binärobjekte schreiben und lesen

Mit *Streams* können die Daten blockweise geschrieben bzw. gelesen werden.

`PreparedStatement`-Methode:

```
void setBinaryStream(int parameterIndex, InputStream x, int length)
    throws SQLException
```

`length` ist die Größe der Datei in Byte.

`ResultSet`-Methoden:

```
InputStream getBinaryStream(int columnIndex) throws SQLException
InputStream getBinaryStream(String columnName) throws SQLException
```

Programm 2.10

Programm 2.10 speichert ein jpeg-Bild für eine bestimmte ISBN-Nummer. Die Bilder sind in einem vorgegebenen Verzeichnis zu finden und tragen jeweils den Dateinamen: *<isbn>.jpeg*.

Aufrufbeispiel:

```
java -cp build ImportImage bilder 3-15-001308-9
```

ImportImage

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class ImportImage {
  public static void main(String[] args) {
    String dir = args[0];
    String isbn = args[1];

    InputStream fin = null;
    Connection con = null;
    try {
      FileInputStream in = new FileInputStream(
         "dbconnect.properties");
      Properties prop = new Properties();
      prop.load(in);
      in.close();

      String driver = prop.getProperty("driver");
      String url = prop.getProperty("url");
```

```
      String user = prop.getProperty("user");
      String password = prop.getProperty("password");

      Class.forName(driver);
      con = DriverManager.getConnection(url, user, password);

      String image = dir + "/" + isbn + ".jpeg";
      File file = new File(image);
      int length = (int) file.length();
      fin = new FileInputStream(file);

      PreparedStatement ps = con.prepareStatement(
        "update buch set bild = ? where isbn = ?");

      ps.setBinaryStream(1, fin, length);
      ps.setString(2, isbn);

      int count = ps.executeUpdate();
      if (count > 0)
        System.out.println(image + " wurde importiert");

      ps.close();
    }
    catch (Exception e) {
      System.err.println(e);
    }
    finally {
      try {
        if (con != null)
          con.close();
        if (fin != null)
          fin.close();
      }
      catch (Exception e) {
        System.err.println(e);
      }
    }
  }
}
```

Programm 2.11 Mit Programm 2.11 kann ein solches gespeichertes jpeg-Bild wieder in ein Dateiverzeichnis exportiert werden.

ExportImage

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class ExportImage {
  public static void main(String[] args) {
    String dir = args[0];
    String isbn = args[1];

    InputStream fin = null;
    FileOutputStream fout = null;
    Connection con = null;
```

```
    try {
      FileInputStream in = new FileInputStream(
         "dbconnect.properties");
      Properties prop = new Properties();
      prop.load(in);
      in.close();

      String driver = prop.getProperty("driver");
      String url = prop.getProperty("url");
      String user = prop.getProperty("user");
      String password = prop.getProperty("password");

      Class.forName(driver);
      con = DriverManager.getConnection(url, user, password);

      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(
        "select bild from buch where isbn = '" + isbn +
        "' and bild is not null");

      if (rs.next()) {
        fin = rs.getBinaryStream(1);
        String image = dir + "/" + isbn + ".jpeg";
        fout = new FileOutputStream(image);
        byte[] buffer = new byte[1024];
        int size;
        while ((size = fin.read(buffer)) != -1) {
          fout.write(buffer, 0, size);
        }
        fout.flush();
        System.out.println("Bild wurde exportiert: " + image);
      }
      else
        System.out.println("Kein Bild vorhanden");

      rs.close();
      stmt.close();
    }
    catch (Exception e) {
      System.out.println(e);
    }
    finally {
      try {
        if (con != null)
          con.close();
        if (fin != null)
          fin.close();
        if (fout != null)
          fout.close();
      }
      catch (Exception e) {
        System.err.println(e);
      }
    }
  }
}
```

2.7 Navigation und Änderungen in der Ergebnismenge

Bei der Erzeugung von SELECT-Anweisungen kann auf die Art der Verwendung der Ergebnismenge Einfluss genommen werden. Hierzu existieren die beiden `Connection`-Methoden

```
Statement createStatement(int resultSetType, int resultSetConcurrency)
    throws SQLException
PreparedStatement prepareStatement(String sql, int resultSetType,
    int resultSetConcurrency) throws SQLException
```

Der Parameter `resultSetType` bestimmt, wie in der Ergebnismenge mit Hilfe eines internen Cursors navigiert werden kann. Der Parameter `resultSetConcurrency` legt fest, ob Datensätze der Ergebnismenge direkt geändert werden können.

Navigierbarkeit

Für `resultSetType` können folgende Konstanten verwendet werden:

`ResultSet.TYPE_FORWARD_ONLY`
Der Cursor kann nur zum nächsten Satz bewegt werden.

`ResultSet.TYPE_SCROLL_INSENSITIVE`
Der Cursor kann frei bewegt werden.

`ResultSet.TYPE_SCROLL_SENSITIVE`
Der Cursor kann frei bewegt werden. Änderungen durch andere Transaktionen sind sichtbar.

Änderbarkeit

Für `resultSetConcurrency` können folgende Konstanten verwendet werden:

`ResultSet.CONCUR_READ_ONLY`
Änderungen können nicht vorgenommen werden.

`ResultSet.CONCUR_UPDATABLE`
Datensätze können in der Ergebnismenge geändert werden.

Die freie Navigation kann bei entsprechender Einstellung mit den folgenden `ResultSet`-Methoden erfolgen:

```
boolean next()
boolean previous()
boolean first()
boolean last()
boolean absolute(int row)
boolean relative(int rows)
void beforeFirst()
void beforeLast()
```

Der Cursor wird zur nächsten, vorherigen, ersten, letzten Zeile bewegt, auf die Zeile mit der Nummer `row` positioniert (die Zäh-

lung beginnt bei 1), um `rows` Zeilen vorwärts bzw. rückwärts (bei negativer Zahl) bewegt, vor die erste Zeile, hinter die letzte Zeile bewegt.

Ferner existieren die Prüfmethoden:

```
boolean isFirst()
boolean isLast()
boolean isBeforeFirst()
boolean isAfterLast()
```

```
int getRow()
```

liefert die Nummer des aktuellen Datensatzes (1 für den ersten Satz).

Programm 2.12 demonstriert die verschiedenen Möglichkeiten der Navigation durch die Ergebnismenge. *Programm 2.12*

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class Navigation {
  public static void main(String[] args) {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
      FileInputStream in = new FileInputStream(
         "dbconnect.properties");
      Properties prop = new Properties();
      prop.load(in);
      in.close();

      String driver = prop.getProperty("driver");
      String url = prop.getProperty("url");
      String user = prop.getProperty("user");
      String password = prop.getProperty("password");

      Class.forName(driver);
      con = DriverManager.getConnection(url, user, password);

      stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
         ResultSet.CONCUR_READ_ONLY);
      rs = stmt.executeQuery(
         "select isbn, titel from buch order by isbn");

      if (rs.last()) {
        System.out.println("Anzahl Zeilen: " + rs.getRow());
      }
      else {
        System.out.println("Ergebnismenge ist leer");
        return;
      }
      rs.beforeFirst();
```

```
      System.out.println(
        "Ihre Eingabe bitte: next, previous, first, last oder quit");

      BufferedReader br = new BufferedReader(
        new InputStreamReader(System.in));
      String line;

      while ((line = br.readLine()) != null) {
        try {
          if (line.equals("next")) {
            if (rs.next())
              System.out.println(rs.getRow() + " " + rs.getString(1)
                + " " + rs.getString(2));
          }
          else if (line.equals("previous")) {
            if (rs.previous())
              System.out.println(rs.getRow() + " " + rs.getString(1)
                + " " + rs.getString(2));
          }
          else if (line.equals("first")) {
            if (rs.first())
              System.out.println(rs.getRow() + " " + rs.getString(1)
                + " " + rs.getString(2));
          }
          else if (line.equals("last")) {
            if (rs.last())
              System.out.println(rs.getRow() + " " + rs.getString(1)
                + " " + rs.getString(2));
          }
          else if (line.equals("quit")) {
            break;
          }
        }
        catch (SQLException e) {
          System.err.println(e);
        }
      }
    }
    catch (Exception e) {
      System.err.println(e);
    }
    finally {
      try {
        if (rs != null)
          rs.close();
        if (stmt != null)
          stmt.close();
        if (con != null)
          con.close();
      }
      catch (SQLException e) { }
    }
  }
}
```

Ist die Ergebnismenge als änderbar eingestellt, können Datensätze direkt in einem `ResultSet`-Objekt eingefügt, geändert und gelöscht werden. In der Regel darf die SELECT-Abfrage nur eine Tabelle umfassen (keine JOINs), keine Aggregat-Funktion, GROUP-Klausel und ORDER BY-Klausel enthalten und muss den Primärschlüssel mit einschließen. Die genauen Regeln hängen vom DBMS ab.

Für das Folgende setzen wir voraus, dass der Cursor auf einem bestimmten Datensatz der Ergebnismenge positioniert ist.

Ändern

Aufruf der `ResultSet`-Methoden

```
void updateXxx(int idx, typ x) throws SQLException
```

analog zu den `setXxx`-Methoden aus Kapitel 2.4.3, also z. B.

```
rs.updateInt(3, 100);
```

Aufruf der `ResultSet`-Methode

```
void updateRow() throws SQLException
```

Einfügen

Aufruf der `ResultSet`-Methode

```
void moveToInsertRow() throws SQLException
```

Aufruf der `updateXxx`-Methoden.

Aufruf der `ResultSet`-Methode

```
void insertRow() throws SQLException
```

Ggf. Aufruf der `ResultSet`-Methode

```
void moveToCurrentRow() throws SQLException
```

um zum aktuellen Datensatz zurückzukehren.

Löschen

Aufruf der `ResultSet`-Methode

```
void deleteRow() throws SQLException
```

Programm 2.13

Programm 2.13 demonstriert die drei Änderungsarten. Zu einem Buch kann die Bestandszahl geändert werden, ein neues Buch mit ISBN-Nummer kann eingefügt werden, ein Datensatz kann gelöscht werden.

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class Update {
  public static void main(String[] args) {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
```

```
try {
  FileInputStream in = new FileInputStream(
     "dbconnect.properties");
  Properties prop = new Properties();
  prop.load(in);
  in.close();

  String driver = prop.getProperty("driver");
  String url = prop.getProperty("url");
  String user = prop.getProperty("user");
  String password = prop.getProperty("password");

  Class.forName(driver);
  con = DriverManager.getConnection(url, user, password);

  stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
     ResultSet.CONCUR_UPDATABLE);
  rs = stmt.executeQuery(
    "select isbn, titel, bestand, stand from buch");

  if (rs.last()) {
    System.out.println("Anzahl Zeilen: " + rs.getRow());
  }
  else {
    System.out.println("Ergebnismenge ist leer");
    return;
  }
  rs.first();
  System.out.println();
  System.out.println(rs.getRow() + " " + rs.getString(1)
    + " " + rs.getString(2) + " " + rs.getString(3));
  System.out.println();

  System.out.println("Ihre Eingabe bitte: ");
  System.out.println(
    "<Bestand>, insert <ISBN>, delete, nur RETURN oder quit");

  BufferedReader br = new BufferedReader(
    new InputStreamReader(System.in));
  String line;

  while ((line = br.readLine()) != null) {
    if (line.equals("quit"))
      break;

    try {
      if (line.equals("")) {
        if (rs.next())
          System.out.println(rs.getRow() + " " + rs.getString(1)
            + " " + rs.getString(2) + " " + rs.getString(3));
        else {
          System.out.println("ENDE");
          break;
        }
      }
      else if (line.startsWith("insert ")) {
        String isbn = line.substring(7).trim();
```

```
            rs.moveToInsertRow();
            rs.updateString(1, isbn);
            rs.insertRow();
            rs.moveToCurrentRow();
            System.out.println("insert OK");
          }
          else if (line.equals("delete")) {
            rs.deleteRow();
            System.out.println("delete OK");
            rs.previous();
          }
          else {
            try {
              int bestand = Integer.parseInt(line);
              rs.updateInt(3, bestand);
              rs.updateTimestamp(4, new Timestamp(
                System.currentTimeMillis()));
              rs.updateRow();
              System.out.println(rs.getRow() + " " + rs.getString(1)
                + " " + rs.getString(2) + " " + rs.getString(3));
            }
            catch (NumberFormatException e) {
              System.err.println("Keine Integerzahl");
            }
          }
        }
        catch (SQLException e) {
          System.err.println(e);
        }
      }
    }
    catch (Exception e) {
      System.err.println(e);
    }
    finally {
      try {
        if (rs != null)
          rs.close();
        if (stmt != null)
          stmt.close();
        if (con != null)
          con.close();
      }
      catch (SQLException e) { }
    }
  }
}
```

2.8 Exkurs: XML-Dokumente aus SQL-Abfragen erzeugen

Thema dieses Kapitels ist die Entwicklung eines Programms, das das Ergebnis einer beliebigen SQL-Abfrage in eine XML-Struktur überführt und diese dann in ein anderes Format (z.B. HTML) mittels XSLT (Extensible Stylesheet Language Transformation) transformiert.

XML

Zunächst einige Bemerkungen zu XML:

Mit so genannten *Auszeichnungssprachen* (wie z.B. HTML) können beliebigen Textelementen Eigenschaften zugewiesen werden, wodurch deren Formatierung oder Bedeutung festgelegt wird. Die Textelemente werden durch Markierungen (*tags*) gekennzeichnet. Auf eine Startmarkierung folgt das Textelement, das in der Regel durch eine Endemarkierung abgeschlossen wird. Dabei sind die Markierungen im Allgemeinen geschachtelt, d.h. im markierten Text können weitere Markierungen enthalten sein. Das ermöglicht eine hierarchische Strukturierung des Inhalts.

XML (Extensible Markup Language) ermöglicht mit selbst definierten Tags eine genaue Beschreibung strukturierter Informationen, die dann maschinell ausgewertet werden können. Im Unterschied zur Sprache HTML, deren Tags fest vorgegeben sind, ist XML eine *Metasprache*, mit der anwendungsspezifische Auszeichnungssprachen definiert werden können.

Programm 2.14

Das XML-Dokument *param.xml* enthält Angaben zur Herstellung einer Datenbankverbindung, eine SQL-Abfrage und einige Dateinamen.

Der allgemeine Aufbau dieser Datei sieht wie folgt aus:

```
<params>
  <param>
    <param-name>Parametername</param-name>
    <param-value>Parameterwert</param-value>
  </param>
  ...
</params>
```

Das zu entwickelnde Programm 2.14 (`DB2XML`) liest diese Parameter mit Hilfe der Klasse `XMLParameters` zu Beginn ein.

param.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<params>
  <!-- MySQL -->
  <param>
    <param-name>driver</param-name>
    <param-value>com.mysql.jdbc.Driver</param-value>
  </param>
  <param>
    <param-name>url</param-name>
    <param-value>jdbc:mysql://localhost/buecher</param-value>
  </param>
  <param>
    <param-name>user</param-name>
    <param-value>root</param-value>
  </param>
  <param>
    <param-name>password</param-name>
    <param-value>root</param-value>
  </param>

  <!-- SQL-Abfrage -->
  <param>
    <param-name>sql</param-name>
    <param-value>
      select autor, titel, isbn, preis, bestand, stand
      from buch order by autor, titel
    </param-value>
  </param>

  <!-- XML-Ausgabe -->
  <param>
    <param-name>xmlFile</param-name>
    <param-value>result/result.xml</param-value>
  </param>

  <!-- XSLT-Template -->
  <param>
    <param-name>xslFile</param-name>
    <param-value>template.xsl</param-value>
  </param>

  <!-- Output -->
  <param>
    <param-name>outputFile</param-name>
    <param-value>result/result.html</param-value>
  </param>
</params>
```

Die Leistung der Klasse `XMLParameters` besteht im Parsen des XML-Dokuments mit Hilfe eines *SAX-Parsers* (Simple API for XML) und der Bereitstellung der Parameternamen und -werte in Form eines `Properties`-Objekts.

Ein SAX-Parser bietet ein ereignisorientiertes API. Der Parser liest das XML-Dokument sequentiell durch und ruft spezielle *Call-*

back-Methoden immer dann auf, wenn er im Eingabestrom ein Tag oder ein Textelement entdeckt.

XMLParameters

```
package xmlutils;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;
import org.xml.sax.XMLReader;
import org.xml.sax.InputSource;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
import java.io.*;
import java.util.*;

public class XMLParameters extends DefaultHandler {
  private String element;
  private String name = "";
  private String value = "";
  private Properties properties = new Properties();

  public XMLParameters(String filename) throws FileNotFoundException,
    IOException, ParserConfigurationException, SAXException {

    SAXParserFactory factory = SAXParserFactory.newInstance();
    SAXParser parser = factory.newSAXParser();
    XMLReader reader = parser.getXMLReader();
    reader.setContentHandler(this);
    reader.parse(new InputSource(new FileReader(filename)));
  }

  public void startElement(String namespaceURI, String localName,
    String qName, Attributes attrs) throws SAXException {
    element = qName;
  }

  public void characters(char[] ch, int start, int length)
    throws SAXException {

    // Textinhalte können sich über mehrere Zeilen erstrecken.

    String text = new String(ch, start, length).trim();
    if (text.length() == 0)
      return;

    if (element.equals("param-name")) {
      if (name.length() > 0)
        name += " ";
      name += text;
    }
    else if (element.equals("param-value")) {
      if (value.length() > 0)
        value += " ";
      value += text;
    }
  }
```

```
  public void endElement(String namespaceURI, String localName,
    String qName) throws SAXException {

    if (qName.equals("param")) {
      properties.setProperty(name, value);
      name = "";
      value = "";
    }
  }

  public Properties getParameters() {
    return properties;
  }
}
```

`XMLParameters` ist von der Klasse `DefaultHandler` abgeleitet, die Default-Implementierungen für alle Callback-Methoden enthält. Die Callback-Methoden `startElement`, `endElement` und `characters` werden in der Subklasse `XMLParameters` überschrieben.

Im Konstruktor von `XMLParameters` wird ein `SAXParserFactory`-Objekt erzeugt, mit dessen Hilfe dann der eigentliche Parser erzeugt wird. Die Methode `parse` des Interface `XMLReader` liest das XML-Dokument und ruft Callback-Methoden auf.

Die Callback-Methoden in `XMLParameters` sind so implementiert, dass führende und nachfolgende Leerzeichen beim Parameternamen und -wert ignoriert werden. Insbesondere kann sich der Parameterwert über mehrere Zeilen erstrecken.

Die Methode `convert` der Klasse `XMLConverter` erzeugt aus dem Ergebnis einer SQL-Abfrage (`ResultSet`) ein XML-Dokument. der Form:

```
<result>
  <headers>
    <header>Spaltenname1</header>
    ...
  </headers>
  <rows>
    <row>
      <Spaltenname1>Wert1</Spaltenname1>
      ...
    </row>
    ...
  </rows>
</result>
```

Die kritischen Zeichen `&`, `<`, `>`, `'` und `"` werden durch so genannte *Entity-Referenzen* ersetzt.

XMLConverter

```
package xmlutils;
import java.sql.*;

public class XMLConverter {
  private ResultSet rs;

  public XMLConverter(ResultSet rs) {
    this.rs = rs;
  }

  public String convert() throws SQLException {
    StringBuilder sb = new StringBuilder();
    String sep = System.getProperty("line.separator");

    sb.append("<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>"
      + sep);
    sb.append("<result>" + sep);
    sb.append("\t<headers>" + sep);

    ResultSetMetaData rsmd = rs.getMetaData();
    int n = rsmd.getColumnCount();
    for (int i = 1; i <= n; i++) {
      sb.append("\t\t<header>" + rsmd.getColumnLabel(i)
        + "</header>" + sep);
    }

    sb.append("\t</headers>" + sep);
    sb.append("\t<rows>" + sep);

    while (rs.next()) {
      sb.append("\t\t<row>" + sep);
      for (int i = 1; i <= n; i++) {
        sb.append("\t\t\t<" + rsmd.getColumnLabel(i) + ">" +
          replace(rs.getString(i)) +
          "</" + rsmd.getColumnLabel(i) + ">" + sep);
      }
      sb.append("\t\t</row>" + sep);
    }

    sb.append("\t</rows>" + sep);
    sb.append("</result>");

    return sb.toString();
  }

  private String replace(String value) {
    StringBuilder sb = new StringBuilder();
    int n = value.length();
    for (int i = 0; i < n; i++) {
      char c = value.charAt(i);
      switch (c) {
        case '&':   sb.append("&amp;");
              break;
        case '<':   sb.append("&lt;");
              break;
        case '>':   sb.append("&gt;");
              break;
```

```
        case '\'':  sb.append("&apos;");
              break;
        case '\"':  sb.append("&quot;");
              break;
        default:  sb.append(c);
      }
    }
    return sb.toString();
  }
}
```

Die Methode `transform` der Klasse `XMLTransformer` wandelt ein XML-Dokument in eine andere Struktur (z.B. ein HTML-Dokument) mit Hilfe eines Stylesheets um. Die *Extensible Stylesheet Language Transformation* (XSLT) beschreibt, wie XML-Dokumente in andere Formate transformiert werden können.

In unserem Beispiel wird das Stylesheet *template.xsl* verwendet.

template.xsl

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" encoding="ISO-8859-1"/>
<xsl:template match="result">
  <html>
    <head>
      <title>SQL-Ergebnis</title>
    </head>
    <body>
      <table border="1" cellpadding="5">
        <tr bgcolor="lightGrey">
          <xsl:apply-templates select="headers/header"/>
        </tr>
        <xsl:apply-templates select="rows/row"/>
      </table>
    </body>
  </html>
</xsl:template>

<xsl:template match="header">
  <th align="left">
    <font face="Arial"><xsl:value-of select="."/></font>
  </th>
</xsl:template>

<xsl:template match="row">
  <tr>
    <xsl:apply-templates select="*"/>
  </tr>
</xsl:template>
```

```
<xsl:template match="*">
  <td>
    <font face="Arial"><xsl:value-of select="."/></font>
  </td>
</xsl:template>
</xsl:stylesheet>
```

template-Elemente legen fest, wie die Ausgabe aussehen soll. Ihr Inhalt definiert eine Vorlage, die mit den Daten des XML-Dokuments gefüllt wird. Das Attribut `match` des Tags `xsl:template` gibt an, für welches Element des XML-Dokuments diese Vorlage gilt.

XMLTransformer

```
package xmlutils;

import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.Transformer;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;

import java.io.*;

public class XMLTransformer {
  private Transformer transformer;

  public XMLTransformer(Reader xslInput)
    throws TransformerConfigurationException {

    TransformerFactory factory = TransformerFactory.newInstance();
    transformer = factory.newTransformer(new StreamSource(xslInput));
  }

  public String transform(Reader xmlInput)
    throws TransformerException {
    StringWriter out = new StringWriter();
    transformer.transform(new StreamSource(xmlInput),
      new StreamResult(out));
    return out.toString();
  }
}
```

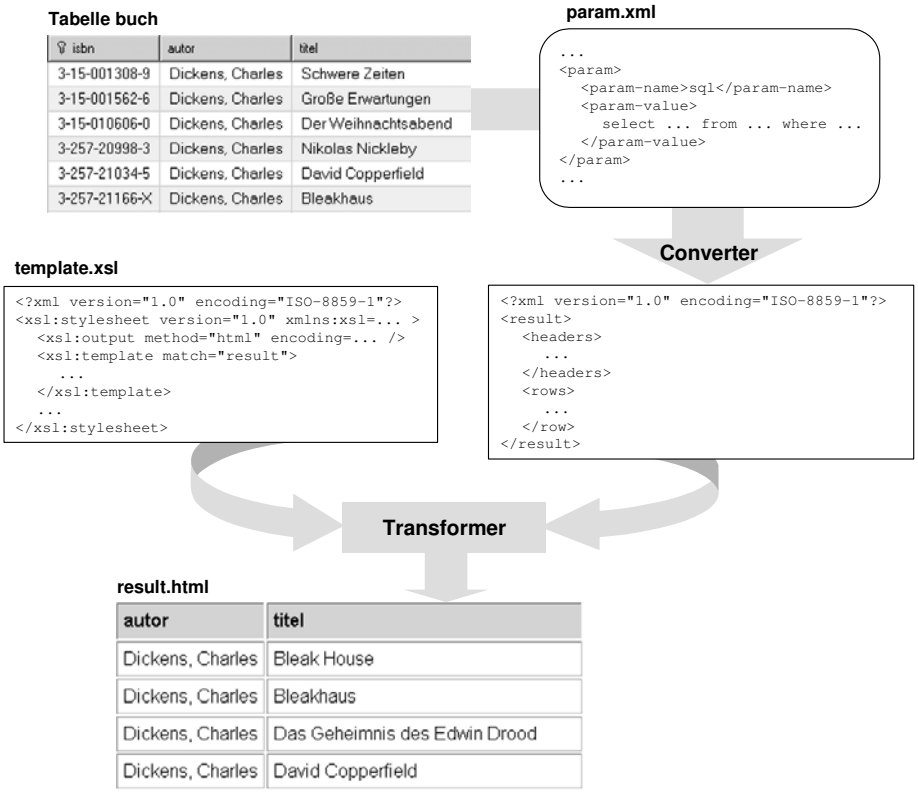
Im Konstruktor von `XMLTransformer` wird ein `TransformerFactory`-Objekt erzeugt, mit dessen Hilfe dann der eigentliche Transformer auf der Basis eines Stylesheets erzeugt wird. Die Methode `transform` der Klasse `Transformer` wandelt das XML-Dokument um.

Die `main`-Methode der Klasse `DB2XML` führt die folgenden Schritte aus:

- Einlesen der Parameter aus *param.xml* mit Hilfe von `XML-Parameters`

- Aufbau der Datenbankverbindung
- Ausführung der SQL-Abfrage
- Erzeugung des XML-Dokuments mit Hilfe von `XMLConverter`
- XSL-Transformation auf Basis des Stylesheets *template.xsl* mit Hilfe von `XMLTransformer`
- Ausgabe in *result.html*

***Bild 2.11**: Programmablauf*



DB2XML

```
import java.sql.*;
import java.io.*;
import java.util.*;
import xmlutils.*;

public class DB2XML {
  public static void main(String[] args) {
    Properties prop = null;
    Connection con = null;
    PrintWriter xmlOut = null;
    PrintWriter out = null;
```

```
try {
  // Parameter einlesen
  XMLParameters params = new XMLParameters("param.xml");
  prop = params.getParameters();

  String driver = prop.getProperty("driver");
  String url = prop.getProperty("url");
  String user = prop.getProperty("user");
  String password = prop.getProperty("password");

  // Treiber laden
  Class.forName(driver);

  // Verbindung zur DB herstellen
  con = DriverManager.getConnection(url, user, password);

  // SQL-Abfrage ausführen
  Statement stmt = con.createStatement();
  String sql = prop.getProperty("sql");
  if (sql == null)
    throw new Exception("Parameter \"sql\" fehlt");
  ResultSet rs = stmt.executeQuery(sql);

  // XML-Ausgabe erzeugen
  XMLConverter conv = new XMLConverter(rs);
  String xml = conv.convert();

  rs.close();
  stmt.close();

  String xmlFile = prop.getProperty("xmlFile");
  if (xmlFile != null) {
    xmlOut = new PrintWriter(new FileWriter(xmlFile));
    xmlOut.print(xml);
    xmlOut.flush();
    System.out.println("XML-Ausgabe: " + xmlFile);
  }

  // XSL-Transformation
  String xslFile = prop.getProperty("xslFile");
  if (xslFile == null)
    throw new Exception("Parameter \"xslFile\" fehlt");
  XMLTransformer trans = new XMLTransformer(
     new FileReader(xslFile));
  String output = trans.transform(new StringReader(xml));

  String outputFile = prop.getProperty("outputFile");
  if (outputFile == null)
    throw new Exception("Parameter \"outputFile\" fehlt");
  out = new PrintWriter(new FileWriter(outputFile));
  out.print(output);
  out.flush();
  System.out.println("Ausgabe der XSL-Transformation: " +
    outputFile);
}
```

```
    catch (Exception e) {
      System.err.println(e);
    }
    finally {
      try {
        if (con != null)
          con.close();
        if (xmlOut != null)
          xmlOut.close();
        if (out != null)
          out.close();
      }
      catch (Exception e) {
        System.err.println(e);
      }
    }
  }
}
```

XML-Dokument result.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<result>
  <headers>
    <header>autor</header>
    <header>titel</header>
    <header>isbn</header>
    <header>preis</header>
    <header>bestand</header>
    <header>stand</header>
  </headers>
  <rows>
    <row>
      <autor>Dickens, Charles</autor>
      <titel>Bleak House</titel>
      <isbn>3-458-32810-6</isbn>
      <preis>17</preis>
      <bestand>16</bestand>
      <stand>2006-08-20 00:00:00.0</stand>
    </row>
    ...
  </rows>
</result>
```

2.9 Exkurs: Stored Procedures

Stored Procedures (gespeicherte Prozeduren) bestehen aus SQL-Anweisungen und zusätzlichen Befehlen zur Deklaration von Variablen, zur Bildung von Schleifen, Verzweigungen usw.

Vorteile und Nachteile

Solche Prozeduren ermöglichen es, einen Teil der Logik vom Clientprogramm auf den Datenbankserver zu verlagern, was oft zu einer Performanceverbesserung führt, da die Prozedur komplett auf dem Datenbankserver stattfindet und keine Übertragung von Zwischenergebnissen zum Client erfolgen muss. Durch die Auslagerung zentraler Codeteile auf den Datenbankserver kön-

nen Coderedundanzen in den Anwendungsprogrammen vermieden werden, was die Anwendungen besser wartbar macht. Beispielsweise muss bei Änderung des Datenmodells in der Datenbank nur eine Stored Procedure geändert werden, nicht aber der Code in allen betroffenen Anwendungsprogrammen.

Leider ist die Definition von Stored Procedures abhängig vom jeweiligen Datenbankmanagementsystem. Fast jedes System nutzt eine eigene Syntax und zusätzliche Spracherweiterungen, was eine Portierung auf ein anderes System sehr aufwändig machen kann.

In den folgenden Beispielen nutzen wir das DBMS MySQL, das ab Version 5.0 Stored Procedures untersützt.

CallableStatement

Das Interface `java.sql.CallableStatement`, ein Subinterface von `java.sql.PreparedStatement`, wird zum Aufruf von Stored Procedures eingesetzt.

Programm 2.15

Im ersten Beispiel nutzen wir die Stored Procedure *zeigeBuecher*, die alle Bücher aus der Tabelle *buch* (siehe Kapitel 2.2.1) liefert, deren Erscheinungsjahr größer oder gleich einem als Parameter vorgegebenen Jahr ist.

Prozedur zeigeBuecher

```
delimiter $$
create procedure zeigeBuecher (in j int)
begin
  select isbn, autor, titel from buch where jahr >= j;
end $$
delimiter ;
```

Die Prozedur hat einen mit `IN` gekennzeichneten Eingabeparameter. Da der Code selbst das Zeichen `;` enthält, ist dieses als Delimiter-Zeichen zum Abschließen von SQL-Anweisungen nicht geeignet und wird deshalb temporär in das Zeichen `$$` geändert.

Die Prozedur kann beispielsweise mit dem Kommandointerpreter *mysql* erstellt werden:

```
mysql -u root -p buecher < zeigeBuecher.sql
```

ZeigeBuecher

Die folgende JDBC-Anwendung ruft die Stored Procedure mit einem `int`-Parameter auf.

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class ZeigeBuecher {
  public static void main(String[] args) throws Exception {
    int jahr = Integer.parseInt(args[0]);
```

```
    FileInputStream in = new FileInputStream("dbconnect.properties");
    Properties prop = new Properties();
    prop.load(in);
    in.close();

    String driver = prop.getProperty("driver");
    String url = prop.getProperty("url");
    String user = prop.getProperty("user");
    String password = prop.getProperty("password");

    Class.forName(driver);
    Connection con = DriverManager.getConnection(url, user, password);
    String str = "{call zeigeBuecher (?)}";
    CallableStatement stmt = con.prepareCall(str);
    stmt.setInt(1, jahr);

    ResultSet rs = stmt.executeQuery();
    while (rs.next()) {
      System.out.println(rs.getString(1) + " " + rs.getString(2)
         + " " + rs.getString(3));
    }

    rs.close();
    stmt.close();
    con.close();
  }
}
```

Die `Connection`-Methode

`CallableStatement` **`prepareCall`**`(String sql) throws SQLException`

erzeugt ein `CallableStatement`-Objekt. Der Aufruf der Prozedur wird in Escape-Syntax notiert, wobei die Parameter jeweils durch `?` festgelegt werden. So enthält `sql` im obigen Beispiel die Zeichenkette: `{call zeigeBuecher (?)}`.

Zur Belegung der `IN`-Parameter stehen alle `setXxx`-Methoden des Interface `PreparedStatement` zur Verfügung.

Mit `executeQuery` wird die Ausführung der Prozedur gestartet und das Ergebnis der Abfrage zurückgeliefert.

In einer Stored Procedure können Parameter zur Eingabe (`IN`), zur Ausgabe (`OUT`) oder zur Ein- und Ausgabe (`INOUT`) benutzt werden. Dies zeigt das folgende Beispiel: ***Programm 2.16***

Prozedur updateBestand

```
delimiter $$
create procedure updateBestand(in buchnr varchar(13), in menge int,
  out code int, out alt int, out neu int)
begin
  declare cnt int;

  main: begin
    select count(*) from buch where isbn = buchnr into cnt;
```

```
    if cnt = 0 then
      set code = 1;
      leave main;
    end if;

    select bestand from buch where isbn = buchnr into alt;
    set neu = alt + menge;
    update buch set bestand = neu where isbn = buchnr;
    set code = 0;
  end main;
end $$
delimiter ;
```

Die Prozedur erhöht bzw. vermindert den Lagerbestand eines Buches (*buchnr*) um eine vorgegebene Menge (*menge*). Ausgabeparameter sind *code* (0 = erfolgreiche Ausführung, 1 = Buch wurde nicht gefunden), der Bestand vor der Änderung (`alt`) und der Bestand nach der Änderung (`neu`).

UpdateBestand

```
import java.sql.*;
import java.io.*;
import java.util.*;

public class UpdateBestand {
  public static void main(String[] args) throws Exception {
    String isbn = args[0];
    int menge = Integer.parseInt(args[1]);

    FileInputStream in = new FileInputStream("dbconnect.properties");
    Properties prop = new Properties();
    prop.load(in);
    in.close();

    String driver = prop.getProperty("driver");
    String url = prop.getProperty("url");
    String user = prop.getProperty("user");
    String password = prop.getProperty("password");

    Class.forName(driver);
    Connection con = DriverManager.getConnection(url, user, password);
    String str = "{call updateBestand (?,?,?,?,?)}";
    CallableStatement stmt = con.prepareCall(str);
    stmt.setString(1, isbn);
    stmt.setInt(2, menge);
    stmt.registerOutParameter(3, Types.INTEGER);
    stmt.registerOutParameter(4, Types.INTEGER);
    stmt.registerOutParameter(5, Types.INTEGER);

    stmt.executeUpdate();
    int rc = stmt.getInt(3);
    if (rc == 0) {
      int alt = stmt.getInt(4);
      int neu = stmt.getInt(5);
```

```
      System.out.println("Bestand alt: " + alt);
      System.out.println("Bestand neu: " + neu);
    }
    else {
      System.out.println("Buch nicht vorhanden");
    }

    stmt.close();
    con.close();
  }
}
```

Ausgabeparameter (OUT) sowie Ein-/Ausgabeparameter (INOUT) müssen registriert werden. Hierzu steht die folgende CallableStatement-Methode zur Verfügung:

```
void registerOutParameter(int idx, int sqlType) throws SQLException
```

idx bestimmt die Position des Parameters (die Zählung beginnt bei 1), sqlType bestimmt den JDBC-Typ (siehe Bild 2.8).

Da die obige Prozedur eine Änderungsanweisung enthält, wird executeUpdate verwendet. Zum Auslesen der Ausgabeparameter werden dem JDBC-Typ entsprechende getXxx-Methoden des Interfaces CallableStatement benutzt.

2.10 Aufgaben

1. Das Datenmodell in Kapitel 2.2.1 soll um *Kunden*, *Bestellungen* und *Bestellpositionen* ergänzt werden:

   Tabelle *kunde*:

kunde_id	integer
kname	varchar(30)
strasse	varchar(20)
plz	varchar(5)
ort	varchar(20)

   Tabelle *bestellung*:

bestell_id	siehe unten
kunde_id	integer
datum	date
status	integer

   Tabelle *bestellposition*:

bestell_id	integer
isbn	varchar(17)
menge	integer

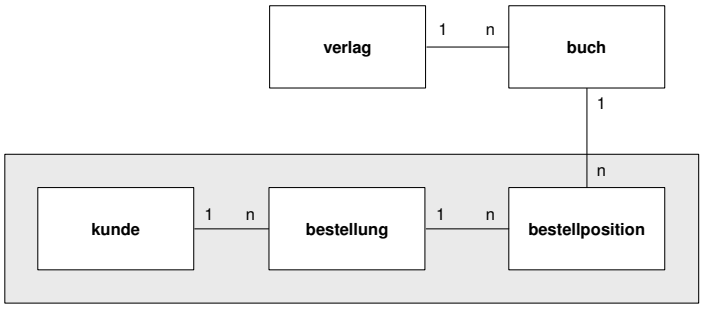


***Bild 2.12**: Erweitertes Datenmodell*

Die *bestell_id* in der Tabelle *bestellung* soll mit Unterstützung des DBMS beim INSERT als laufende Nummer vergeben werden. Dazu gibt es bei den verschiedenen Systemen unterschiedliche Lösungen.

Bei Access bewirkt der Datentyp `counter` für die Spalte *bestell_id* diesen Mechanismus.

Bei *MySQL* kann in der CREATE-TABLE-Anweisung *auto_increment* genutzt werden:

```
bestell_id integer auto_increment
```

Für *Apache Derby* sieht die entsprechende Anweisung so aus:

```
bestell_id integer generated always as identity
```

Nutzen Sie Programm 2.5, um die Tabellen einzurichten und fügen Sie einige Kundendaten in die Tabelle *kunde* ein.

2. Schreiben Sie ein Programm, das für einen Kunden Bestelldaten aufnimmt. Das Programm soll mit den Parametern *kunde_id* und *datei* aufgerufen werden. Die Kundennummer muss eine gültige ID aus Tabelle *kunde* sein. *datei* bezeichnet eine Datei, die z.B. folgenden Inhalt hat:

```
3-15-001308-9   5
3-15-001562-6   3
3-15-010606-0   8
```

Pro Zeile ist die ISBN-Nummer des bestellten Buches und die gewünschte Stückzahl enthalten.

Die Spalte *status* soll den Wert 0 erhalten.

Die erzeugte Bestellnummer in der Tabelle *bestellung* muss in der Tabelle *bestellposition* als Fremdschlüssel eingetragen werden. Für die einzelnen Datenbanksysteme gibt es dazu unterschiedliche Verfahren:

bestellung

bestell_id	kunde_id	datum	status
1	1	2006-08-22	0

bestellposition

bestell_id	isbn	menge
1	3-15-001308-9	5
1	3-15-001562-6	3
1	3-15-010606-0	8

***Bild 2.13**: Ergebnisbeispiel*

Access

Nach "`insert into bestellung ...`" kann die letzte automatisch generierte Nummer mit der Anweisung "`select @@identity`" abgefragt werden. Diese wird dann in die Tabelle *bestellposition* eingetragen.

MySQL und Apache Derby

Wird an die `Statement`-Methode `executeUpdate` als zweiter Parameter der Wert `Statment.RETURN_GENERATED_KEYS` übergeben, so liefert die `Statement`-Methode `getGeneratedKeys()` ein `ResultSet`-Objekt, das die generierte Nummer enthält.

3. Bestellte Bücher müssen für die Auslieferung vorbereitet werden. Schreiben Sie ein Programm, das für eine vorgegebene *bestell_id* folgende Aktionen durchführt:

   a) Prüfen, ob die beim Programmaufruf mitgegebene ID als *bestell_id* in der Tabelle *bestellung* vorkommt.

   Wenn ja:

   b) Prüfen, ob diese Bestellung noch unbearbeitet (Status = 0) ist.

   Wenn ja:

   c) Verfügbarkeit prüfen.
   Pro Bestellposition muss geprüft werden, ob die gewünschte Stückzahl die Bestandszahl aus der Tabelle *buch* nicht übersteigt. Entsprechende Meldungen sind auszugeben.

   Wenn *alle* bestellten Bücher in der gewünschten Stückzahl vorrätig sind:

   d) Lagerbestand in der Tabelle *buch* für alle Bestellpositionen gemäß der bestellten Stückzahl reduzieren und den Status in der Tabelle *bestellung* auf den Wert 1 setzen.

4. Entwickeln Sie ein Programm, das die Bestandsdaten (*isbn*, *bestand*, *stand*) der Tabelle *buch* der Bücher-Datenbank (z.B. verwaltet mit *MS Access*) in eine Datenbank eines anderen Datenbanksystems (z.B. *MySQL*) kopiert. Die hierfür

geeignete Tabelle der Zieldatenbank soll ebenfalls mit diesem Programm erstellt werden.

5. Erstellen Sie eine Variante von Programm 2.6, die *Batch-Updates* (siehe Kapitel 2.4.2) benutzt.

3 Verbindungslose Kommunikation mit UDP

Das *User Datagram Protocol* (UDP) stellt grundlegende Funktionen zur Verfügung, um mit geringem Aufwand Daten zwischen kommunizierenden Prozessen austauschen zu können. UDP ist als Transportprotokoll der dritten Schicht im TCP/IP-Schichtenmodell zugeordnet und nutzt den Vermittlungsdienst IP.

3.1 Das Protokoll UDP

UDP ist ein *verbindungsloses Protokoll*, d.h. es bietet keinerlei Kontrolle über die sichere Ankunft versendeter Datenpakete, so genannter *Datagramme*.

UDP ist im RFC (Request for Comments) 768 der IETF (Internet Engineering Task Force) spezifiziert (siehe auch `http://www.ietf.org/rfc/rfc768.txt`).

Datagramm

Datagramme müssen nicht den Empfänger in der Reihenfolge erreichen, in der sie abgeschickt werden. Datagramme können verloren gehen, weder Sender noch Empfänger werden über den Verlust informiert. Fehlerhafte Datagramme werden nicht nochmals übertragen.

Bild 3.1 zeigt den Aufbau des *UDP-Datagramms*, das in ein IPv4-Paket eingebettet ist.

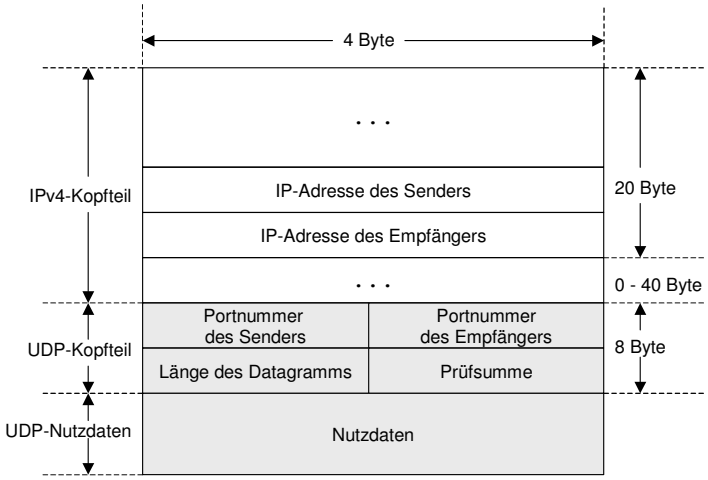


***Bild 3.1**: Aufbau des UDP-Datagramms*

Der *UDP-Kopfteil* enthält vier 16 Bit lange Felder:

- die Portnummer des Senders,
- die Portnummer des Empfängers,
- die Gesamtlänge des UDP-Datagramms und
- die Prüfsumme.

Darauf folgen dann die eigentlichen *Nutzdaten* des Datagramms.

Die Maximallänge eines UDP-Datagramms ist begrenzt. Anwendungen sollten nicht mehr als 508 Byte Nutzdaten in einem Datagramm versenden, um Probleme infolge einer evtl. nötigen Fragmentierung des IP-Pakets durch Router zu vermeiden.

UDP ist ein einfaches Mittel, um Nachrichten, die keine zuverlässige Übertragung erfordern, zu versenden. UDP hat gegenüber TCP den Vorteil eines viel geringeren Kontroll-Overheads (8 Byte bei UDP, 20 Byte bei TCP).

UDP-Anwendungen

Einige bekannte Anwendungen, die UDP nutzen, sind:

- das *Domain Name System* (DNS) zur automatischen Konvertierung von Domain-Namen (Rechnernamen) in IP-Adressen und umgekehrt,
- das *Simple Network Management Protocol* (SNMP) zur Verwaltung von Ressourcen in einem TCP/IP-basierten Netz,
- das *Trivial File Transfer Protocol* (TFTP), das einen einfachen Dateitransferdienst zum Sichern und Laden von System- und Konfigurationsdateien bietet,
- das *Dynamic Host Configuration Protocol* (DHCP) zur dynamischen Zuweisung von IP-Adressen.

Die unidirektionale Übertragung von Video- und Audiosequenzen zur Wiedergabe in Echtzeit (*Multimedia Streaming*) ist besonders zeitkritisch. Der gelegentliche Verlust eines Datenpakets kann aber toleriert werden. Hier ist UDP sehr gut geeignet.

3.2 DatagramSocket und DatagramPacket

Java-Programme nutzen so genannte *Sockets* als Programmierschnittstelle für die Kommunikation über UDP/IP und TCP/IP. Sockets stellen die Endpunkte einer Kommunikationsbeziehung zwischen Prozessen dar.

Ein *UDP-Socket* wird an eine Portnummer auf dem lokalen Rechner gebunden. Er kann dann Datagramme an jeden beliebigen anderen UDP-Socket senden und von jedem beliebigen UDP-Socket empfangen.

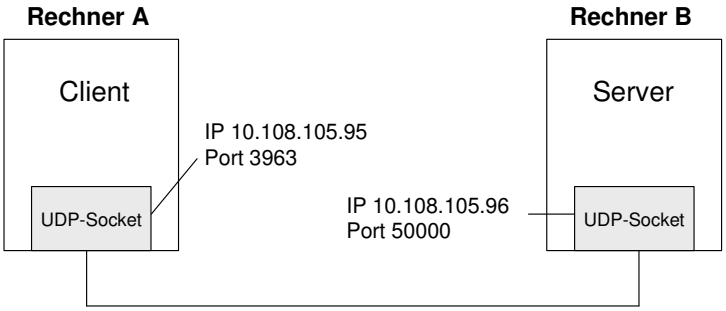


***Bild 3.2**: UDP-Sockets*

DatagramSocket

Die Klasse `java.net.DatagramSocket` repräsentiert einen UDP-Socket zum Senden und Empfangen von UDP-Datagrammen.

Ein UDP-Socket muss an einen Port des lokalen Rechners gebunden werden. Hierzu stehen Konstruktoren zur Verfügung:

`DatagramSocket``() throws SocketException`
erzeugt einen UDP-Socket und bindet ihn an einen verfügbaren Port des lokalen Rechners.

`DatagramSocket``(int port) throws SocketException`
erzeugt einen UDP-Socket und bindet ihn an die Portnummer `port` des lokalen Rechners. Hierdurch kann ein Server den Port, den er nutzen möchte, festlegen.

Beide Konstruktoren können bei Fehlern im zugrunde liegenden Protokoll die Ausnahme `java.net.SocketException` auslösen:

`SocketException` ist Subklasse von `java.io.IOException`.

Die beiden Methoden

`InetAddress` **`getLocalAddress`**`()` und
`int` **`getLocalPort`**`()`

liefern die IP-Adresse bzw. die Portnummer, an die der UDP-Socket gebunden ist.

`void` **`close`**`()`
schließt den UDP-Socket.

DatagramPacket

Die Klasse `java.net.DatagramPacket` repräsentiert ein UDP-Datagramm.

Der Konstruktor

`DatagramPacket``(byte[] buf, int length, InetAddress address, int port)`

erzeugt ein UDP-Datagramm zum *Senden* von Daten. `buf` enthält die Daten, `address` ist die IP-Adresse und `port` die Portnummer des Empfängers. Die Anzahl `length` der zu übertragenden Bytes muss kleiner oder gleich der Pufferlänge sein.

Der Konstruktor

```
DatagramPacket(byte[] buf, int length)
```

erzeugt ein UDP-Datagramm zum *Empfangen* von Daten. `buf` ist der Puffer, der die empfangenen Daten aufnimmt. `length` legt fest, wie viele Bytes maximal gelesen werden sollen. `length` muss kleiner oder gleich der Pufferlänge sein. Wendet man auf dieses Paket die Methoden `setAdress` und `setPort` (siehe weiter unten) an, so kann es auch gesendet werden.

send

Die `DatagramSocket`-Methode

```
void send(DatagramPacket p) throws IOException
```

sendet ein UDP-Datagramm von diesem Socket aus.

receive

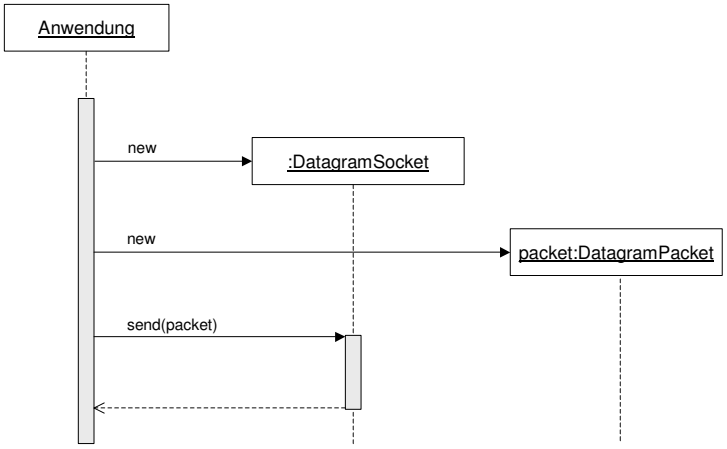
Die `DatagramSocket`-Methode

```
void receive(DatagramPacket p) throws IOException
```

empfängt ein UDP-Datagramm von diesem Socket. `p` enthält die Daten sowie die IP-Adresse und die Portnummer des Senders. Die Methode blockiert so lange, bis ein Datagramm empfangen wurde.

Bild 3.3 zeigt den Ablauf zum Senden eines Datagramms in Form eines UML-Sequenzdiagramms.

***Bild 3.3**: Datagramm senden*



Weitere Datagram-Packet-Methoden

Weitere `DatagramPacket`-Methoden sind:

`byte[] **getData**()`
liefert den Datenpufferinhalt.

`int **getLength**()`
liefert die Länge der Daten, die empfangen wurden bzw. gesendet werden sollen.

`InetAddress **getAddress**()`
liefert die (entfernte) IP-Adresse des Senders (beim empfangenen Datagramm) bzw. des Empfängers (beim zu sendenden Datagramm).

`int **getPort**()`
liefert die (entfernte) Portnummer des Senders (beim empfangenen Datagramm) bzw. des Empfängers (beim zu sendenden Datagramm).

`void **setData**(byte[] buf)`
setzt den Datenpuffer des Datagramms.

`void **setLength**(int length)`
bestimmt die Anzahl Bytes, die gesendet werden sollen bzw. die maximale Anzahl Bytes, die empfangen werden sollen. `length` muss kleiner oder gleich der Pufferlänge des Datagramms sein.

`void **setAddress**(InetAddress address)`
setzt die IP-Adresse des Rechners, an den das Datagramm gesendet werden soll.

`void **setPort**(int port)`
setzt die Portnummer, an die das Datagramm gesendet werden soll.

In allen Beispielen dieses Kapitels ist die Bearbeitungszeit des Servers nach Eintreffen eines Pakets relativ kurz. Deshalb ist eine *iterative Implementierung* (Paket empfangen – Anfrage bearbeiten – Paket senden) ausreichend.

Bei längeren Bearbeitungszeiten sollten für die Bearbeitung und das Senden der Antwort zu einer Anfrage Threads benutzt werden (*parallele Implementierung*), um die durchschnittliche Wartezeit eines Clients zu reduzieren.

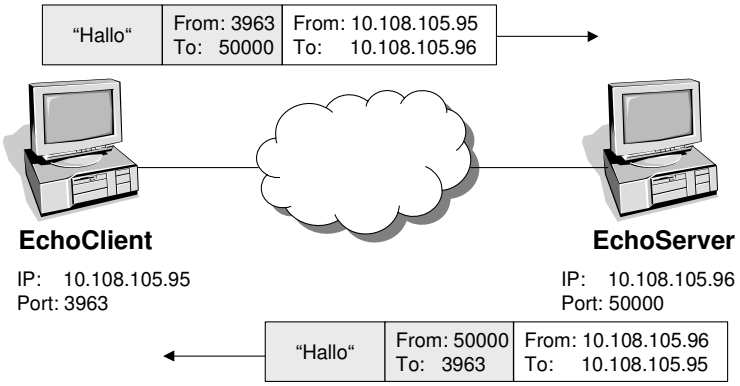
3.3 Ein Echo-Server und -Client

Das erste Programmbeispiel ist eine einfache Client-Server-Anwendung, die den Umgang mit den Methoden des vorigen Abschnitts zeigen soll.

Programm 3.1

Der Client schickt einen Text, den der Server als Echo an den Absender zurückschickt.

Bild 3.4: *Echo-Dienst*



EchoServer

```
import java.net.*;

public class EchoServer {
  private static final int BUFSIZE = 508;

  public static void main(String[] args) throws Exception {
    int port = Integer.parseInt(args[0]);

    // Socket an Port binden
    DatagramSocket socket = new DatagramSocket(port);

    // Pakete zum Empfangen bzw. Senden
    DatagramPacket packetIn = new DatagramPacket(
      new byte[BUFSIZE], BUFSIZE);
    DatagramPacket packetOut = new DatagramPacket(
      new byte[BUFSIZE], BUFSIZE);

    while (true) {
      // Paket empfangen
      socket.receive(packetIn);
      System.out.print(".");

      // Daten und Länge im Antwortpaket speichern
      packetOut.setData(packetIn.getData());
      packetOut.setLength(packetIn.getLength());
```

```
      // Zieladresse und Zielport im Antwortpaket setzen
      packetOut.setAddress(packetIn.getAddress());
      packetOut.setPort(packetIn.getPort());

      // Antwortpaket senden
      socket.send(packetOut);
    }
  }
}
```

`EchoServer` wird mit dem Parameter *Portnummer* aufgerufen.

Zu Beginn wird ein UDP-Socket erzeugt und an die vorgegebene Portnummer gebunden. Es werden zwei Datagramme zum Empfangen bzw. Senden von Daten mit der Pufferlänge 508 Byte erzeugt. In einer Endlos-Schleife werden Daten empfangen und gesendet. Nutzdaten und Datenlänge sowie IP-Adresse und Portnummer werden aus dem empfangenen Paket gelesen und in das zu sendende Paket übertragen.

Der Server kann über Tastatur mit `Strg+C` abgebrochen werden.

EchoClient

Das Programm `EchoClient` sendet ein Datagramm an den Server. Neben dem Hostnamen und der Portnummer des Servers werden die Nutzdaten des Datagramms als weiterer Parameter beim Aufruf mitgegeben.

Erhält der Client nicht innerhalb von zwei Sekunden eine Antwort vom Server, wird eine Ausnahme ausgelöst und das Programm beendet.

Diese *Zeitsteuerung* wird durch Aufruf der folgenden `DatagramSocket`-Methode erreicht:

`void` **`setSoTimeout`**`(int timeout) throws SocketException`
setzt ein Timeout in Millisekunden. `receive` für diesen Socket blockiert höchstens `timeout` Millisekunden und löst dann die Ausnahme `java.net.SocketTimeoutException` aus. Wird `timeout = 0` gesetzt, so wird die *Timeout-Steuerung* deaktiviert. `SocketTimeoutException` ist Subklasse von `java.io.InterruptedIOException`.

`int` **`getSoTimeout`**`() throws SocketException`
liefert die Timeout-Angabe.

```
import java.net.*;

public class EchoClient {
  private static final int BUFSIZE = 508;
  private static final int TIMEOUT = 2000;

  public static void main(String[] args) throws Exception {
    try {
      String host = args[0];
      int port = Integer.parseInt(args[1]);
```

```
      byte[] data = args[2].getBytes();

      // Socket wird an anonymen Port gebunden
      DatagramSocket socket = new DatagramSocket();

      // Maximal TIMEOUT Millisekunden auf Antwort warten
      socket.setSoTimeout(TIMEOUT);

      // Paket an Server senden
      InetAddress addr = InetAddress.getByName(host);
      DatagramPacket packetOut = new DatagramPacket(
        data, data.length, addr, port);
      socket.send(packetOut);

      // Antwortpaket empfangen
      DatagramPacket packetIn = new DatagramPacket(
         new byte[BUFSIZE], BUFSIZE);
      socket.receive(packetIn);

      String received = new String(
         packetIn.getData(), 0, packetIn.getLength());
      System.out.println(received);

      socket.close();
    }
    catch (SocketTimeoutException e) {
      System.err.println("Timeout: " + e.getMessage());
    }
  }
}
```

3.4 Einschränkung der Verbindungen

Die Klasse `DatagramSocket` enthält eine Methode, die es ermöglicht, die Adresse, an die Datagramme gesendet werden bzw. von der Datagramme empfangen werden, gezielt auszuwählen und alle anderen Datagramme zu verwerfen:

`void` **`connect`**`(InetAddress address, int port)`
Datagramme können nur an diese IP-Adresse/Portnummer gesendet bzw. von dieser IP-Adresse/Portnummer empfangen werden.

`void` **`disconnect`**`()`
hebt die durch `connect` hergestellte Restriktion auf.

Mit den beiden `DatagramSocket`-Methoden

`InetAddress` **`getInetAddress`**`()` und
`int` **`getPort`**`()`

können die IP-Adresse bzw. die Portnummer, die in `connect` verwendet wurden, abgefragt werden. Wurde `connect` nicht benutzt, wird `null` bzw. `-1` geliefert.

Programm 3.2

Der Client *Sender* schickt kurze Texte, die über Tastatur in einer Eingabeschleife erfasst werden, als Datagramme an den Server. Der Server *Receiver* zeigt die empfangenen Daten auf dem Bildschirm an. Er soll nur Datagramme, die von einer bestimmten IP-Adresse und Portnummer aus gesendet wurden, empfangen können. Zu diesem Zweck wird der Server mit drei Parametern aufgerufen: *lokale Portnummer*, *IP-Adresse/Rechnername des Client* und *Portnummer des Client*.

Receiver

```
import java.net.*;

public class Receiver {
  private static final int BUFSIZE = 508;

  public static void main(String[] args) throws Exception {
    int port = Integer.parseInt(args[0]);
    String remoteHost = args[1];
    int remotePort = Integer.parseInt(args[2]);

    DatagramSocket socket = new DatagramSocket(port);

    // Es werden nur Datagramme dieser Adresse entgegengenommen
    socket.connect(InetAddress.getByName(remoteHost), remotePort);

    DatagramPacket packet = new DatagramPacket(
      new byte[BUFSIZE], BUFSIZE);

    while (true) {
      socket.receive(packet);
      String text = new String(
         packet.getData(), 0, packet.getLength());
      System.out.println(packet.getAddress().getHostAddress() + ":" +
        packet.getPort());
      System.out.println(text);
      System.out.println();
    }
  }
}
```

Beim Client *Sender* wird der UDP-Socket an eine explizit vorgegebene Portnummer gebunden, die beim Aufruf des Programms als Parameter mitgegeben wird.

Sender

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

public class Sender {
  private static final int BUFSIZE = 508;

  public static void main(String[] args) throws Exception {
    int localPort = Integer.parseInt(args[0]);
```

```
    String host = args[1];
    int port = Integer.parseInt(args[2]);
    // Socket wird an vorgegebenen Port gebunden
    DatagramSocket socket = new DatagramSocket(localPort);

    InetAddress addr = InetAddress.getByName(host);
    DatagramPacket packet = new DatagramPacket(
      new byte[BUFSIZE], BUFSIZE, addr, port);

    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));

    while (true) {
      String text = in.readLine();
      if (text.equals("."))
        break;

      byte[] data = text.getBytes();
      packet.setData(data);
      packet.setLength(data.length);
      socket.send(packet);
    }

    in.close();
    socket.close();
  }
}
```

Aufrufbeispiel

Aufruf des Serverprogramms:

```
start java -cp build Receiver 50000 localhost 40000
```

Es können nur Datagramme von `localhost` mit Portnummer `40000` empfangen werden.

Aufruf des Clientprogramms auf demselben Rechner:

```
java -cp build Sender 40000 localhost 50000
```

Der Socket des Clients ist an die Portnummer `40000` gebunden. Wird hier beim Aufruf statt `40000` z. B. `30000` angegeben, so ignoriert der Server die vom Client gesendeten Datagramme.

3.5 Online-Unterhaltung

Mit dem folgenden Programm *Talk* können zwei Benutzer an verschiedenen Rechnern eine Unterhaltung online führen.

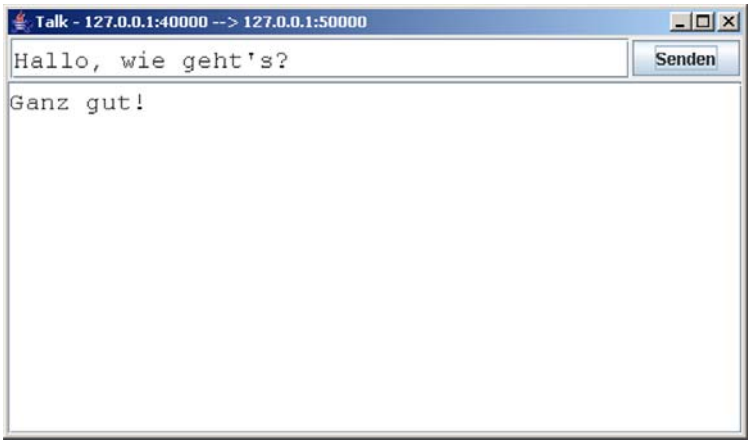


***Bild 3.5**: Online-Unterhaltung*

Szenario

Benutzer 1 sitzt am Rechner A (IP-Adresse 10.108.105.95), Benutzer 2 am Rechner B (IP-Adresse 10.108.105.96).

Benutzer 1 ruft das Programm wie folgt auf:

```
java -cp build Talk 50000 10.108.105.96 50000
```

Benutzer 2 gibt das Folgende ein:

```
java -cp build Talk 50000 10.108.105.95 50000
```

Der UDP-Socket wird hier jeweils an die explizit vorgegebene lokale Portnummer `50000` (erster Parameter) gebunden.

Zum Test kann das Programm auch zweimal auf demselben Rechner (`localhost`) gestartet werden. Allerdings müssen dann die Portnummern unterschiedlich sein:

```
java -cp build Talk 40000 localhost 50000
```

```
java -cp build Talk 50000 localhost 40000
```

Programm 3.3

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.net.*;
import java.io.*;

public class Talk implements ActionListener, Runnable {
  private static final int BUFSIZE = 508;
  private DatagramSocket socket;
  private DatagramPacket packetOut;
  private JTextField input;
  private JTextArea output;

  public static void main(String[] args) {
    if (args.length != 3) {
      System.err.println(
        "java Talk <localPort> <remoteHost> <remotePort>");
      System.exit(1);
    }

    JFrame frame = new JFrame();

    try {
      new Talk(args, frame);
    }
    catch (Exception e) {
      System.err.println(e);
      System.exit(1);
    }

    frame.pack();
    frame.setVisible(true);
  }

  public Talk(String[] args, JFrame frame) throws Exception {
    int localPort = Integer.parseInt(args[0]);
    String remoteHost = args[1];
    int remotePort = Integer.parseInt(args[2]);

    InetAddress remoteAddr = InetAddress.getByName(remoteHost);

    frame.setTitle("Talk - " +
      InetAddress.getLocalHost().getHostAddress()
      + ":" + localPort + " --> " +
      remoteAddr.getHostAddress() + ":" + remotePort);

    socket = new DatagramSocket(localPort);
    packetOut = new DatagramPacket(
      new byte[BUFSIZE], BUFSIZE, remoteAddr, remotePort);

    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        if (socket != null)
          socket.close();
        System.exit(0);
      }
    });
```

```
    Container c = frame.getContentPane();

    JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT, 2, 2));
    input = new JTextField(40);
    input.setFont(new Font("Monospaced", Font.PLAIN, 18));
    panel.add(input);
    JButton send = new JButton("Senden");
    send.addActionListener(this);
    panel.add(send);
    c.add(panel, BorderLayout.NORTH);

    output = new JTextArea(10, 45);
    output.setFont(new Font("Monospaced", Font.PLAIN, 18));
    output.setLineWrap(true);
    output.setWrapStyleWord(true);
    output.setEditable(false);
    c.add(new JScrollPane(output), BorderLayout.CENTER);

    // Thread zum Empfangen von Nachrichten starten
    Thread t = new Thread(this);
    t.start();
  }

  public void run() {
    DatagramPacket packetIn = new DatagramPacket(
      new byte[BUFSIZE], BUFSIZE);

    while (true) {
      try {
        receive(packetIn);
      }
      catch (IOException e) {
          break;
      }
    }
  }

  private void receive(DatagramPacket packetIn) throws IOException {
    socket.receive(packetIn);
    final String text = new String(
      packetIn.getData(), 0, packetIn.getLength());

    try {
      EventQueue.invokeLater(new Runnable() {
        public void run() {
          output.append(text);
          output.append("\n");
        }
      });
    }
    catch (Exception e) { }
  }

  public void actionPerformed(ActionEvent e) {
    try {
      byte[] data = input.getText().getBytes();
      packetOut.setData(data);
```

```
        packetOut.setLength(data.length);
        socket.send(packetOut);
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
```

Im Konstruktor werden zunächst IP-Adressen bestimmt, der UDP-Socket an die vorgegebene lokale Portnummer gebunden und ein Paket zum Senden der eingegebenen Daten erstellt. Beim Schließen des Fensters wird der Socket geschlossen. Textfeld, Button und Textfläche werden erzeugt und platziert sowie das aktuelle Objekt als `ActionListener` beim Button registriert. Schließlich wird ein Thread, der Datagramme empfängt (siehe `run`-Methode), erzeugt und gestartet.

Innerhalb der `run`-Methode werden in einer Endlos-Schleife die Daten empfangen und in der Textfläche angezeigt. Die `DatagramSocket`-Methode `receive` blockiert so lange, bis ein Datagramm empfangen wurde. Die Ausgabe des Textes erfolgt in der `run`-Methode eines `Runnable`-Objekts `r`, das mittels `EventQueue.invokeLater(r)` an den Ereignis-Dispatcher, der für die Aktualisierung der Benutzungsoberfläche verantwortlich ist, zur Ausführung weitergeleitet wird.

Beim Drücken des Buttons "Senden" werden die Daten des Eingabefeldes in einem Datagramm an die Zieladresse gesendet.

3.6 Aufgaben

1. Entwickeln Sie einen Server (`DaytimeServer`), der als Reaktion auf den Empfang eines "leeren" Datagramms die aktuelle Systemzeit in Form einer Zeichenkette an den Sender zurückschickt.

   Programmieren Sie für den Test auch einen passenden Client (`DaytimeClient`), der maximal zwei Sekunden auf die Antwort des Servers wartet.

2. Ein Client (`MesswertClient`) sendet in Abständen von 5 Sekunden Zufallszahlen mit Zeitpunktangaben an den Server.

   Der Server (`MesswertServer`) gibt diese Zahlen zusammen mit der IP-Adresse des Senders am Bildschirm aus.

Beispiel:

```
10.108.105.95:3975 10:45:46 48.72014957156734
10.108.105.95:3975 10:45:51 48.557565388620226
10.108.105.95:3975 10:45:56 70.66274469656827
```

3. Ein Server (`QuoteServer`) liefert zufallsgesteuert Zitate. Alle Zitate befinden sich in einer Datei *quotes.txt*, die den folgenden Aufbau hat:

   *Zitat (*eine Zeile)
   Autor des Zitats (eine Zeile)
   Zitat (eine Zeile)
   Autor des Zitats (eine Zeile)
   ...

   Erhält der Server ein "leeres" Datagramm, so bestimmt eine Zufallszahl, welches Zitat (mit Autor) als Nächstes an den Client gesendet wird.

   Programmieren Sie zum einen eine Anwendung mit grafischer Oberfläche (`QuoteApp`) und zum andern ein Applet (`QuoteApplet`) zur Anzeige des angeforderten Zitats.

4 Client/Server-Anwendungen mit TCP

Das wichtigste Protokoll der Transportschicht im TCP/IP-Schichtenmodell ist das *Transmission Control Protocol* (TCP). Es ist aufwändiger als UDP, stellt aber dafür eine verlässliche Verbindung zwischen Client und Server her. Viele bekannte Internet-Dienste wie *FTP* (File Transfer Protocol), *Telnet*, *SMTP* (Simple Mail Transfer Protocol), *POP* (Post Office Protocol) und *HTTP* (Hypertext Transfer Protocol) nutzen TCP.

4.1 Das Protokoll TCP

TCP ist im Gegensatz zu UDP ein *verbindungsorientiertes Protokoll*. Zwischen zwei Prozessen (Client und Server) wird eine virtuelle Verbindung hergestellt, über die in Form eines bidirektionalen Datenstroms Bytefolgen beliebiger Länge geschickt werden können.

Vor der Übertragung von Daten muss der Client eine Verbindung zum Server anfordern. Wird die Verbindung nicht mehr gebraucht, fordert einer der beteiligten Prozesse TCP auf, sie abzubauen.

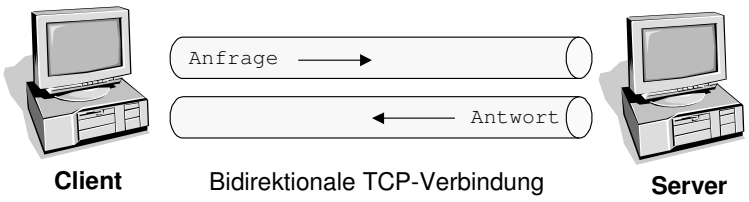


***Bild 4.1**: TCP-Verbindung*

TCP sorgt für die Aufteilung der Daten in einzelne Pakete, garantiert, dass die Pakete den Empfänger in der richtigen Reihenfolge erreichen, und initiiert die Neuübertragung von verloren gegangenen oder defekten Paketen.

TCP ist im RFC 793 der IETF spezifiziert (siehe auch `http://www.ietf.org/rfc/rfc793.txt`).

Bild 4.2 zeigt den Aufbau eines TCP-Pakets, das Bestandteil der Nutzdaten des IPv4-Pakets ist.

Der *TCP-Kopfteil* enthält:

- die Portnummer des Senders (2 Byte),
- die Portnummer des Empfängers (2 Byte),

- weitere Felder mit einer Gesamtlänge von 16 Byte wie z.B. Reihenfolgenummer und Prüfsumme,
- optionale Felder variabler Länge.

Darauf folgen dann die eigentlichen *Nutzdaten* des TCP-Pakets.

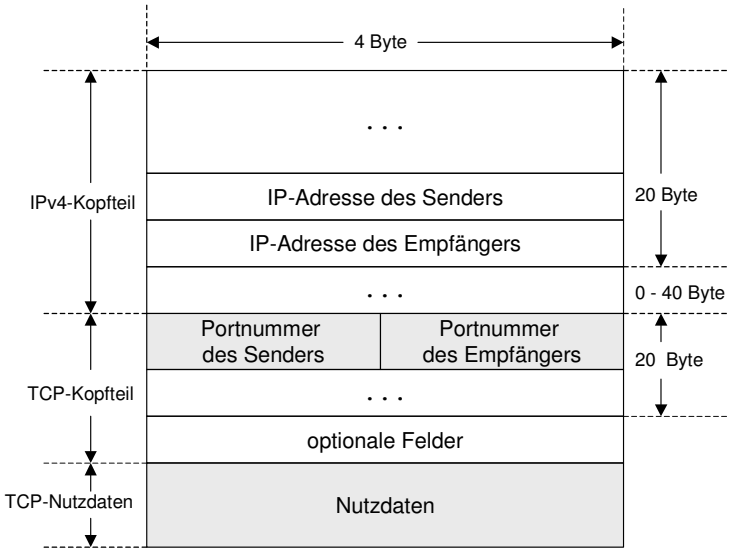


***Bild 4.2**: Aufbau des TCP-Pakets*

4.2 TCP-Sockets

Client-Socket und Server-Socket

Im Gegensatz zu UDP wird bei TCP zwischen *Client-Socket* und *Server-Socket* unterschieden.

Der *Server-Socket* versetzt den Server in die Lage, die Verbindungsanforderungen der Clients abzuhören und dann mehrere Clients zu bedienen.

Bevor Daten zwischen Client und Server ausgetauscht werden können, muss eine Verbindung zwischen Sockets hergestellt werden. Der Aufruf der Methode `accept` des Server-Sockets blockiert den Server so lange, bis ein Client versucht, Kontakt aufzunehmen. `accept` liefert als Ergebnis einen weiteren Socket, über den dann die eigentliche Kommunikation stattfindet.

Die Kontaktaufnahme auf der Client-Seite geschieht automatisch bei der Erzeugung des *Client-Sockets* mit einem geeigneten Konstruktor. Eine Verbindung kann natürlich nur hergestellt werden, wenn der Server zeitlich vor dem Start des Client `accept` aufgerufen hat. Über Ein- und Ausgabeströme können nun Daten empfangen bzw. gesendet werden.

Die Verbindung wird abgebaut, wenn einer der beiden Sockets geschlossen wird.

Socket

Die Klasse `java.net.Socket` implementiert einen *Client-Socket*.

`Socket(InetAddress address, int port) throws IOException`
erzeugt einen Client-Socket und bindet ihn an die vorgegebene IP-Adresse und Portnummer.

`Socket(String host, int port) throws java.net.UnknownHostException, IOException`
erzeugt einen Client-Socket und bindet ihn an den vorgegebenen Rechner und an die vorgegebene Portnummer.

`void close() throws IOException`
schließt den Socket.

`boolean isClosed()`
liefert `true`, falls der Socket geschlossen ist, sonst `false`.

`void setSoTimeout(int timeout) throws SocketException`
setzt ein Timeout in Millisekunden. `read` für den Eingabestrom dieses Sockets blockiert höchstens `timeout` Millisekunden und löst dann die Ausnahme `java.net.SocketTimeoutException` aus. Wird `timeout = 0` gesetzt, so wird die *Timeout-Steuerung* deaktiviert.

`int getSoTimeout() throws SocketException`
liefert die Timeout-Angabe.

Input / Output

`InputStream getInputStream() throws IOException`
liefert einen Eingabestrom für diesen Socket.

`OutputStream getOutputStream() throws IOException`
liefert einen Ausgabestrom für diesen Socket.

Wird einer der beiden Datenströme mit der entsprechenden `close`-Methode geschlossen, so wird auch der zugehörige Socket geschlossen.

Mit einer der folgenden Methoden kann jeweils nur einer der beiden Datenströme (Eingabe bzw. Ausgabe) geschlossen werden, ohne damit den Socket selbst zu schließen:

`void shutdownInput() throws IOException`
setzt den Eingabestrom für diesen Socket auf EOF (End Of File).

`void shutdownOutput() throws IOException`
deaktiviert den Ausgabestrom für diesen Socket.

ServerSocket

Die Klasse `java.net.ServerSocket` implementiert einen *Server-Socket*.

`ServerSocket(int port) throws IOException`
erzeugt einen Server-Socket und bindet ihn an die vorgegebene Portnummer.

`void close() throws IOException`
schließt den Server-Socket.

`boolean isClosed()`
liefert `true`, falls der Server-Socket geschlossen ist, sonst `false`.

`Socket accept() throws IOException`
nimmt einen Verbindungswunsch an und erzeugt einen neuen Socket für diese Verbindung. `accept` blockiert so lange, bis eine neue Verbindung aufgenommen wurde.

`void setSoTimeout(int timeout) throws SocketException`
setzt ein Timeout in Millisekunden. `accept` blockiert höchstens `timeout` Millisekunden und löst dann die Ausnahme `java.net.SocketTimeoutException` aus. Wird `timeout = 0` gesetzt, so wird die *Timeout-Steuerung* deaktiviert.

`int getSoTimeout() throws IOException`
liefert die Timeout-Angabe.

backlog

Verbindungswünsche, für die mittels `accept` noch kein Socket erzeugt werden konnte, werden in einer Warteschlange (*backlog*) verwaltet. Falls die Warteschlange voll ist und ein neuer Verbindungswunsch eingeht, wird beim Client die Ausnahme `java.net.ConnectException` ausgelöst. Die maximale Länge der Warteschlange ist standardmäßig 50.

Mit dem Konstruktor

`ServerSocket(int port, int backlog) throws IOException`

kann die maximale Länge der Warteschlange explizit festgelegt werden.

Ablaufschritte

1. Der Server erzeugt ein `ServerSocket`-Objekt, das an einen vorgegebenen Port gebunden wird.
2. Der Server ruft die Methode `accept` des Server-Sockets auf und wartet auf den Verbindungswunsch eines Client.
3. Der Client erzeugt ein `Socket`-Objekt mit der Adresse und der Portnummer des Servers und versucht damit, eine Verbindung zum Server aufzunehmen.
4. Der Server erzeugt ein `Socket`-Objekt, das das serverseitige Ende der Kommunikationsverbindung darstellt. Die Methode `accept` gibt dieses Objekt zurück.
5. Um eine bidirektionale Verbindung zwischen Client und Server aufzubauen, stellen Client und Server jeweils einen Aus- und Eingabestrom mit Hilfe ihrer `Socket`-Objekte bereit.
6. Nun können Daten mittels Lese- und Schreiboperationen hin- und hergeschickt werden.

Bild 4.3 zeigt den Ablauf beim Aufbau einer TCP-Verbindung.

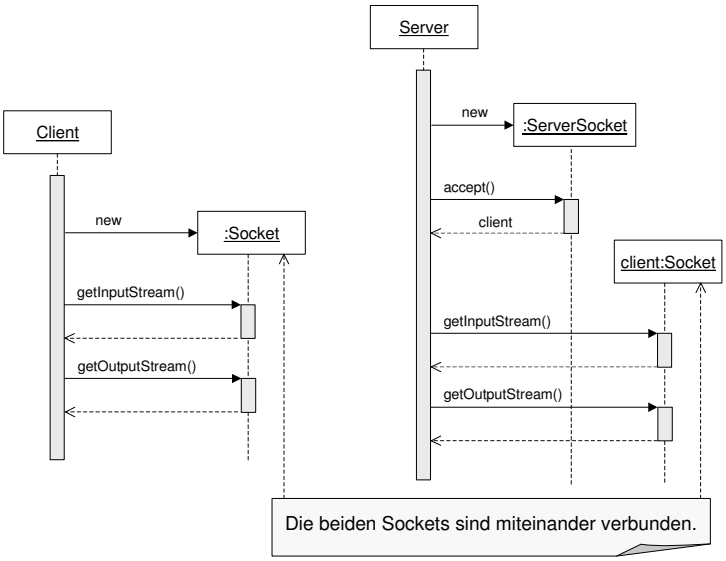


***Bild 4.3**: Aufbau einer TCP-Verbindung*

Socket-Informationen

Weitere `Socket`-Methoden sind:

`InetAddress` **`getLocalAddress`**`()`
liefert die lokale IP-Adresse, an die der Socket gebunden ist.

`int` **`getLocalPort`**`()`
liefert die lokale Portnummer, an die der Socket gebunden ist.

`InetAddress` **`getInetAddress`**`()`
liefert die entfernte IP-Adresse, mit der der Socket verbunden ist.

`int` **`getPort`**`()`
liefert die entfernte Portnummer, mit der der Socket verbunden ist.

4.3 Ein Echo-Server und -Client

Wir stellen nun eine verbindungsorientierte Version des Echo-Dienstes vor.

Programm 4.1

Der Client schickt in einer Schleife jeweils eine Textzeile zum Server, der diese dann wieder an den Client zurückschickt. Die Eingabe erfolgt über Tastatur.

Hier ist zunächst der Client:

EchoClient

Die eigentliche Verarbeitung findet in einer `try`-Anweisung statt. Die Verbindung zum Server wird am Ende mit `socket.close()` abgebaut.

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

public class EchoClient {
  public static void main(String[] args) {
    Socket socket = null;
    try {
      String host = args[0];
      int port = Integer.parseInt(args[1]);
      socket = new Socket(host, port);

      BufferedReader in = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));
      PrintWriter out = new PrintWriter(
         socket.getOutputStream(), true);

      BufferedReader input = new BufferedReader(
        new InputStreamReader(System.in));

      String msg = in.readLine();
      System.out.println(msg);

      String line;
      while (true) {
        line = input.readLine();
        if (line == null || line.equals("q"))
          break;

        out.println(line);
        System.out.println(in.readLine());
      }

      in.close();
      out.close();
      input.close();
    }
    catch (Exception e) {
      System.err.println(e);
    }
    finally {
      try {
        if (socket != null)
          socket.close();
      }
      catch (IOException e) { }
    }
  }
}
```

Zunächst wird ein Client-Socket erzeugt und damit versucht, die Verbindung zum Server herzustellen. Das `BufferedReader`-Objekt `in`

ist der Eingabestrom, das `PrintWriter`-Objekt `out` der Ausgabestrom für diesen Socket.

Dann wird eine Nachricht des Servers ausgegeben. In einer Schleife werden nun von der Tastatur Textzeilen eingelesen, in den Ausgabestrom `out` geschrieben und damit zum Server geschickt. `in.readLine()` blockiert so lange, bis die Antwort vom Server vorliegt. Diese wird dann angezeigt.

Wir zeigen zwei Server-Versionen. Die erste Version kann nicht mehrere Clients gleichzeitig bedienen. Werden z.B. zwei Clients kurz hintereinander gestartet, so werden sie der Reihe nach bedient. Erst wenn der erste Client beendet wurde, kommt der zweite zum Zuge. Es handelt sich also um einen so genannten *iterativen Server*.

EchoServer1

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

public class EchoServer1 {
  private int port;
  private int backlog;

  public EchoServer1(int port, int backlog) {
    this.port = port;
    this.backlog = backlog;
  }

  public void startServer() {
    try {
      ServerSocket server = new ServerSocket(port, backlog);

      InetAddress addr = InetAddress.getLocalHost();
      System.out.println("EchoServer1 auf " +
        addr.getHostName() + "/" + addr.getHostAddress() +
        ":" + port + " gestartet ...");

      process(server);
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }

  private void process(ServerSocket server) throws IOException {
    while (true) {
      Socket client = server.accept();

      String clientAddr = client.getInetAddress().getHostAddress();
      int clientPort = client.getPort();
      System.out.println("Verbindung zu " +
        clientAddr + ":" + clientPort + " aufgebaut");
```

```
      try {
        BufferedReader in = new BufferedReader(
          new InputStreamReader(client.getInputStream()));
        PrintWriter out = new PrintWriter(
          client.getOutputStream(), true);

        out.println("Server ist bereit ...");

        String input;
        while ((input = in.readLine()) != null) {
          out.println(input);
        }

        in.close();
        out.close();
      }
      catch (IOException e) {
        System.err.println(e);
      }
      finally {
        try {
          if (client != null)
            client.close();
        }
        catch (IOException e) { }

        System.out.println("Verbindung zu " +
          clientAddr + ":" + clientPort + " abgebaut");
      }
    }
  }

  public static void main(String[] args) {
    if (args.length != 1 && args.length != 2) {
      System.err.println("java EchoServer1 <port> [<backlog>]");
      System.exit(1);
    }

    int port = Integer.parseInt(args[0]);
    int backlog = 50;
    if (args.length == 2)
      backlog = Integer.parseInt(args[1]);

    new EchoServer1(port, backlog).startServer();
  }
}
```

Der Server-Socket wird erzeugt und an die vorgegebene Portnummer des lokalen Rechners gebunden. Hier wird auch die maximale Länge der Warteschlange (*backlog*) vorgegeben. Somit kann das Verhalten der Anwendung mit unterschiedlichen Werten getestet werden.

Je Schleifendurchgang in der Methode `process` erfolgen diese Schritte:

- Aufruf der Methode `accept`, die so lange blockiert, bis ein Client versucht, Verbindung aufzunehmen.
- Bereitstellung der Ein- und Ausgabeströme `in` und `out`.
- Sendung einer Nachricht an den Client.
- In einer Schleife werden Textzeilen eingelesen und sofort in den Ausgabestrom geschrieben. Die Schleife läuft so lange, bis der Client die Verbindung und damit seinen Ausgabestrom geschlossen hat.
- Der Socket wird geschlossen.

Test

Aufruf des Servers:

```
start java -cp build EchoServer1 50000
```

Aufruf des Client:

```
java -cp build EchoClient localhost 50000
```

Der Server meldet den Auf- und Abbau der Verbindung:

```
EchoServer1 auf pc0806/127.0.0.1:50000 gestartet ...
Verbindung zu 127.0.0.1:2634 aufgebaut
Verbindung zu 127.0.0.1:2634 abgebaut
```

Der Server kann über Tastatur mit `Strg+C` abgebrochen werden.

EchoServer2

Die zweite Version des Echo-Servers ermöglicht die *parallele Bedienung* mehrerer Clients. Dazu erfolgt die Kommunikation mit einem Client innerhalb eines Threads.

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

public class EchoServer2 {
  private int port;

  public EchoServer2(int port) {
    this.port = port;
  }

  public void startServer() {
    try {
      ServerSocket server = new ServerSocket(port);

      InetAddress addr = InetAddress.getLocalHost();
      System.out.println("EchoServer2 auf " +
        addr.getHostName() + "/" + addr.getHostAddress() +
        ":" + port + " gestartet ...");
```

```
    while (true) {
      Socket client = server.accept();
      new EchoThread(client).start();
    }
  }
  catch (IOException e) {
    System.err.println(e);
  }
}

private class EchoThread extends Thread {
  private Socket client;

  public EchoThread(Socket client) {
    this.client = client;
  }

  public void run() {
    String clientAddr = client.getInetAddress().getHostAddress();

    int clientPort = client.getPort();
    System.out.println("Verbindung zu " +
      clientAddr + ":" + clientPort + " aufgebaut");

    try {
      BufferedReader in = new BufferedReader(
        new InputStreamReader(client.getInputStream()));
      PrintWriter out = new PrintWriter(
        client.getOutputStream(), true);

      out.println("Server ist bereit ...");

      String input;
      while ((input = in.readLine()) != null) {
        out.println(input);
      }

      in.close();
      out.close();
    }
    catch (IOException e) {
      System.err.println(e);
    }
    finally {
      try {
        if (client != null)
          client.close();
      }
      catch (IOException e) { }

      System.out.println("Verbindung zu " +
        clientAddr + ":" + clientPort + " abgebaut");
    }
  }
}
```

```
  public static void main(String[] args) {
    int port = Integer.parseInt(args[0]);
    new EchoServer2(port).startServer();
  }
}
```

Sobald die Methode `accept` einen Verbindungswunsch angenommen hat, wird mit Hilfe des gelieferten Sockets ein `EchoThread`-Objekt erzeugt und dieser Thread dann gestartet. Sofort ist der Server wieder bereit, eine neue Verbindung aufzunehmen. Pro Verbindung existiert also ein eigener Thread, der den entsprechenden Client bedient.

4.4 Ein Download-Programm

Client und Server des folgenden Beispiels ermöglichen das Herunterladen von beliebigen Dateien aus einem vorgegebenen Verzeichnis. ***Programm 4.2***

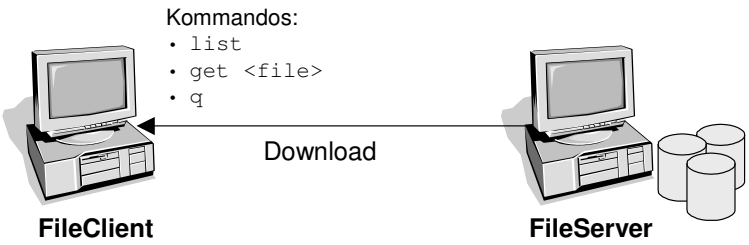


***Bild 4.4**: Download*

Folgende Kommandos kann der Client verwenden:

`list`	Auflistung aller Dateinamen des Serververzeichnisses
`get <file>`	Download der Datei `<file>`
`q`	Beenden des Programms

FileServer

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

public class FileServer {
  private int port;
  private File dir;

  public FileServer(int port, File dir) {
    this.port = port;
    this.dir = dir;
  }
```

```
public void startServer() {
  try {
    ServerSocket server = new ServerSocket(port);

    InetAddress addr = InetAddress.getLocalHost();
    System.out.println("FileServer auf " +
      addr.getHostName() + "/" + addr.getHostAddress() +
      ":" + port + " gestartet ...");

    while (true) {
      Socket client = server.accept();
      new FileThread(client, dir).start();
    }
  }
  catch (IOException e) {
    System.err.println(e);
  }
}

private class FileThread extends Thread {
  private final static long MAX_SIZE = 1024 * 1024;  // 1 MB
  private final static int TIMEOUT = 600000;  // 10 Min.
  private Socket client;
  private File dir;
  private ObjectInputStream in;
  private ObjectOutputStream out;

  public FileThread(Socket client, File dir) {
    this.client = client;
    this.dir = dir;
  }

  public void run() {
    String clientAddr = client.getInetAddress().getHostAddress();
    int clientPort = client.getPort();
    System.out.println("Verbindung zu " +
      clientAddr + ":" + clientPort + " aufgebaut");

    try {
      client.setSoTimeout(TIMEOUT);
      out = new ObjectOutputStream(client.getOutputStream());
      out.flush();
      in = new ObjectInputStream(client.getInputStream());

      while (true) {
        String cmd = (String) in.readObject();
        if (cmd.equals("list")) {
          doList();
        }
        else if (cmd.startsWith("get ")) {
          String file = cmd.substring(4).trim();
          doGet(file);
        }
        else if (cmd.equals("q")) {
          break;
        }
      }
```

```
      in.close();
      out.close();
    }
    catch (EOFException e) { }
    catch (Exception e) {
      System.err.println(e);
    }
    finally {
      try {
        if (client != null)
          client.close();
      }
      catch (IOException e) { }
      System.out.println("Verbindung zu " +
        clientAddr + ":" + clientPort + " abgebaut");
    }
  }

  private void doList() throws IOException {
    // Verzeichnisnamen werden nicht zurückgegeben
    File[] files = dir.listFiles(new FileFilter() {
      public boolean accept(File pathname) {
        if (pathname.isFile())
          return true;
        else
          return false;
      }
    });

    out.writeObject(files);
    out.flush();
  }

  private void doGet(String file) throws IOException {
    File f = new File(dir, file);
    if (!f.isFile()) {
      FileNotFoundException e = new FileNotFoundException(file);
      out.writeObject(e);
      out.flush();
      return;
    }
    if (f.length() > MAX_SIZE) {
      Exception e = new Exception(file + " ist zu gross");
      out.writeObject(e);
      out.flush();
      return;
    }

    BufferedInputStream fin = null;
    try {
      ByteArrayOutputStream bout = new ByteArrayOutputStream();
      fin = new BufferedInputStream(new FileInputStream(f));
      byte[] b = new byte[1024];
      int c;
      while ((c = fin.read(b)) != -1) {
        bout.write(b, 0, c);
      }
```

```
        byte[] bytes = bout.toByteArray();
        out.writeObject(bytes);
      }
      catch (IOException e) {
        System.err.println(e);
      }
      finally {
        try {
          if (fin != null)
            fin.close();
        }
        catch (IOException e) { }
      }
    }
  }

  public static void main(String[] args) {
    if (args.length != 2) {
      System.err.println("java FileServer <port> <dir>");
      System.exit(1);
    }

    int port = Integer.parseInt(args[0]);
    File dir = new File(args[1]);
    if (!dir.isDirectory()) {
      System.err.println(args[1] + " ist kein Verzeichnis");
      System.exit(1);
    }

    new FileServer(port, dir).startServer();
  }
}
```

Der Server wird mit den Parametern *Portnummer* und *Verzeichnis* aufgerufen. Zu Beginn wird geprüft, ob der zweite Parameter ein gültiges Verzeichnis bezeichnet. Die Kommunikation mit dem Client findet in einem clientbezogenen Thread statt (`run`-Methode). Bleibt der Client 10 Minuten inaktiv, wird die Verbindung abgebrochen.

Über den Eingabestrom `in` (vom Typ `ObjectInputStream`) werden die Kommandos (`String`-Objekte) des Client gelesen. Über den Ausgabestrom `out` (vom Typ `ObjectOutputStream`) werden Objekte unterschiedlicher Typen (Dateilisten, Byte-Arrays mit dem Dateiinhalt, Exceptions) an den Client gesendet.

Achtung

Bei der Verwendung von `ObjectInputStream` und `ObjectOutputStream` für die Kommunikation über Sockets ist Folgendes zu beachten:

Der `ObjectOutputStream`-Konstruktor schreibt den so genannten *Serialization Stream Header* in den Ausgabestrom.

Der `ObjectInputStream`-Konstruktor *blockiert*, bis der zugeordnete `ObjectOutputStream` den Header geschrieben hat. Deshalb sollte

unmittelbar nach dem Aufruf des `ObjectOutputStream`-Konstruktors der Puffer mittels `flush()` geleert werden.

Diese Verhaltenweise wird im Client- und Serverprogramm berücksichtigt, ansonsten würden sich die Programme gegenseitig blockieren.

Zum Kommando `list`:
Alle Dateinamen (keine Namen von Unterverzeichnissen) des Serververzeichnisses werden in den Ausgabestrom als Array vom Typ `File` geschrieben.

Zum Kommando `get`:
Wird die angeforderte Datei im Serververzeichnis nicht gefunden oder überschreitet die Dateigröße den Maximalwert `MAX_SIZE`, so wird ein Objekt vom Typ `Exception` in den Ausgabestrom geschrieben. Ansonsten wird die Datei in ein Byte-Array übertragen und dieses dann in den Ausgabestrom geschrieben.

FileClient

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

public class FileClient {
  private String host;
  private int port;
  private File dir;
  private ObjectInputStream in;
  private ObjectOutputStream out;
  private BufferedReader input;

  public FileClient(String host, int port, File dir) {
    this.host = host;
    this.port = port;
    this.dir = dir;
  }

  public void doWork() {
    Socket socket = null;
    try {
      socket = new Socket(host, port);

      in = new ObjectInputStream(socket.getInputStream());
      out = new ObjectOutputStream(socket.getOutputStream());
      out.flush();

      input = new BufferedReader(new InputStreamReader(System.in));

      while (true) {
        System.out.println(
          "Kommando eingeben (list | get <file> | q):");
        String cmd = input.readLine();
```

```
      if (cmd == null || cmd.equals("q")) {
        doQuit();
        break;
      }
      if (cmd.equals("list")) {
        doList();
      }
      else if (cmd.startsWith("get ")) {
        String file = cmd.substring(4).trim();
        doGet(file);
      }
      else
        System.out.println("Ungueltiger Befehl");
    }

    in.close();
    out.close();
    input.close();
  }
  catch (Exception e) {
    System.err.println(e);
  }
  finally {
    try {
      if (socket != null)
        socket.close();
      }
    catch (IOException e) { }
  }
}

private void doQuit() throws IOException {
  out.writeObject("q");
}

private void doList() throws IOException, ClassNotFoundException {
  out.writeObject("list");
  File[] files = (File[]) in.readObject();
  for (File f : files) {
    System.out.println(f.getName());
  }
}

private void doGet(String file) throws IOException,
  ClassNotFoundException {

  out.writeObject("get " + file);
  Object obj = in.readObject();

  if (obj instanceof Exception) {
    System.out.println(obj);
    return;
  }

  BufferedOutputStream fout = null;
```

```
    try {
      File f = new File(dir, file);
      ByteArrayInputStream bin = new ByteArrayInputStream(
         (byte[])obj);
      fout = new BufferedOutputStream(new FileOutputStream(f));
      byte[] b = new byte[1024];
      int c;
      while ((c = bin.read(b)) != -1) {
        fout.write(b, 0, c);
      }
    }
    catch (IOException e) {
      System.err.println(e);
    }
    finally {
      try {
        if (fout != null)
          fout.close();
      }
      catch (IOException e) { }
      System.out.println(file + " wurde uebertragen");
    }
  }

  public static void main(String[] args) {
    if (args.length != 3) {
      System.err.println("java FileClient <host> <port> <dir>");
      System.exit(1);
    }

    String host = args[0];
    int port = Integer.parseInt(args[1]);
    File dir = new File(args[2]);
    if (!dir.isDirectory()) {
      System.err.println(args[2] + " ist kein Verzeichnis");
      System.exit(1);
    }

    new FileClient(host, port, dir).doWork();
  }
}
```

Der Client wird mit den Parametern *Rechnername*, *Portnummer* und *lokales Verzeichnis* aufgerufen. Die übertragenen Dateien werden in dem vorgegebenen Verzeichnis gespeichert.

Zu Beginn wird geprüft, ob der dritte Parameter ein gültiges Verzeichnis bezeichnet.

Über den Ausgabestrom `out` werden die Kommandos gesendet. Über den Eingabestrom `in` werden Objekte gelesen und entsprechend ihrem Typ verarbeitet: Ausgabe einer Liste von Dateinamen, Speicherung des übertragenen Dateiinhalts, Ausgabe der Fehlermeldungen.

4.5 Ein Chat-Programm

Programm 4.3

Mit dem *Chat-Client* ist das so genannte "Chatten" mit mehreren Teilnehmern im Netz möglich. Der Teilnehmer kann sich an- und abmelden und Textzeilen an alle anderen aktiven Teilnehmer senden. Der *Chat-Server* registriert die angemeldeten Teilnehmer und verteilt eingehende Nachrichten an alle registrierten Teilnehmer. Der Chat-Client wird sowohl als Applikation mit grafischer Oberfläche als auch als Applet realisiert.

ChatServer

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

public class ChatServer {
  private static Vector<PrintWriter> manager =
    new Vector<PrintWriter>();
  private int port;

  public ChatServer(int port) {
    this.port = port;
  }

  public void startServer() {
    try {
      ServerSocket server = new ServerSocket(port);

      InetAddress addr = InetAddress.getLocalHost();
      System.out.println("ChatServer auf " +
        addr.getHostName() + "/" + addr.getHostAddress() +
        ":" + port + " gestartet ...");

      while (true) {
        Socket client = server.accept();
        new ChatThread(client).start();
      }
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }

  private class ChatThread extends Thread {
    private final static int TIMEOUT = 600000;  // 10 Min.
    private Socket client;
    private String name;
    private BufferedReader in;
    private PrintWriter out;

    public ChatThread(Socket client) {
      this.client = client;
    }
```

```
public void run() {
  String clientAddr = client.getInetAddress().getHostAddress();
  int clientPort = client.getPort();

  try {
    client.setSoTimeout(TIMEOUT);

    in = new BufferedReader(
      new InputStreamReader(client.getInputStream()));
    out = new PrintWriter(client.getOutputStream(), true);

    login();
    System.out.println("Verbindung zu " +
      clientAddr + ":" + clientPort + " aufgebaut: " + name);

    String message;
    while ((message = in.readLine()) != null) {
      sendMessage(name + ": " + message);
    }

    in.close();
    out.close();
  }
  catch (IOException e) {
    System.err.println(e);
  }
  finally {
    logout();

    try {
      if (client != null)
        client.close();
    }
    catch (IOException e) { }

    System.out.println("Verbindung zu " +
      clientAddr + ":" + clientPort + " abgebaut: " + name);
  }
}

private void login() throws IOException {
  manager.add(out);
  name = in.readLine();
  sendMessage(name + " ist dazugekommen");
}

private void logout() {
  manager.remove(out);
  sendMessage(name + " hat sich verabschiedet");
}
```

```
    private void sendMessage(String message) {
      synchronized (manager) {
        for (PrintWriter out : manager) {
          out.println(message);
        }
      }
    }
  }

  public static void main(String[] args) {
    int port = Integer.parseInt(args[0]);
    new ChatServer(port).startServer();
  }
}
```

Das Programmgerüst entspricht den in den letzten Abschnitten behandelten Beispielen. Kernstück des Programms ist die `run`-Methode.

Der Server verwaltet ein `Vector`-Objekt `manager`, das für alle `Thread`-Objekte dasselbe ist (Klassenvariable). Mit Hilfe dieses Objekts "merkt" sich der Server, wer sich als Teilnehmer angemeldet hat. Nach Aufnahme der Verbindung wird die Referenz `out` auf den Ausgabestrom für diesen Teilnehmer im Vektor gespeichert (siehe Methode `login`) und am Ende aus dem Vektor wieder entfernt (siehe Methode `logout`).

Die erste Textzeile, die der Server über `in` erhält, ist der Login-Name des Teilnehmers. Alle anderen empfangenen Zeilen sind Nachrichten des Teilnehmers an alle aktiven Teilnehmer. Alle Nachrichten werden mit dem Teilnehmernamen versehen und mit der Methode `sendMessage` an alle angemeldeten Teilnehmer gesendet (siehe Bild 4.5). Mit derselben Methode werden die Teilnehmer darüber informiert, wer sich an- oder abgemeldet hat.

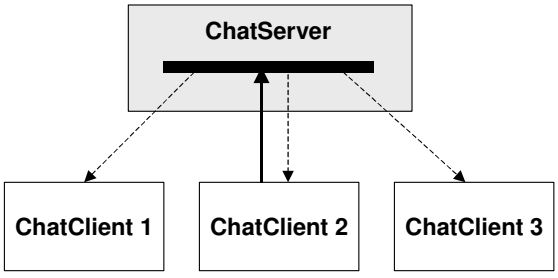


***Bild 4.5**: Der Server als Reflektor*

Der Chat-Client ist so programmiert, dass er als *Applet* oder als *eigenständige Applikation* eingesetzt werden kann. *ChatClient*

Das Einlesen der Nachrichten vom Server erfolgt in einem eigenen Thread (siehe `run`-Methode), der in der Methode `login` gestartet wird.

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ChatClient extends JApplet implements Runnable,
  ActionListener {

  private final static int width = 500;
  private final static int hight = 400;

  private JLabel label;
  private JTextArea area;
  private JTextField text;
  private JButton button;

  private String host;
  private int port;
  private Socket socket;
  private BufferedReader in;
  private BufferedWriter out;

  private volatile Thread t;
  private boolean login = false;
  private String name;

  // Wird verwendet, um den Client als Applikation zu starten.
  private static JFrame frame;

  // Kommandozeilenparameter für die Socket-Initialisierung
  private static String[] cmdLineArgs;

  // Der Client kann auch als Applikation gestartet werden.
  // Dies wird ermöglicht, indem ein Frame für das Applet
  // zur Verfügung gestellt wird.
  static public void main(String[] args) {
    if (args.length != 2) {
      System.err.println("java ChatClient <host> <port>");
      System.exit(1);
    }

    // Wird für Socket-Initialisierung benötigt.
    cmdLineArgs = args;
```

```
  // Der Client
  final ChatClient client = new ChatClient();

  frame = new JFrame("Chat-Client");
  frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      client.destroy();
      System.exit(0);
    }
  });

  client.init();
  frame.getContentPane().add(client);
  frame.setSize(width, hight);
  frame.setVisible(true);
}

public ChatClient() {
  label = new JLabel(" ");
  JPanel top = new JPanel();
  top.add(label);

  area = new JTextArea();
  area.setFont(new Font("Monospaced", Font.PLAIN, 14));
  area.setLineWrap(true);
  area.setEditable(false);

  text = new JTextField(48);
  text.setFont(new Font("Monospaced", Font.PLAIN, 14));
  text.setEnabled(false);
  text.addActionListener(this);

  button = new JButton("Login");
  button.setEnabled(false);
  button.addActionListener(this);

  JPanel input = new JPanel();
  input.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10));
  input.add(text);
  input.add(button);

  Container c = getContentPane();
  c.add(top, BorderLayout.NORTH);
  c.add(new JScrollPane(area), BorderLayout.CENTER);
  c.add(input, BorderLayout.SOUTH);
}

// Verbindungsparameter werden bereitgestellt.
public void init() {
  if (frame == null) {
    host = getCodeBase().getHost();
    port = Integer.parseInt(getParameter("port"));
  }
  else {
    host = cmdLineArgs[0];
    port = Integer.parseInt(cmdLineArgs[1]);
  }
```

```
    text.setEnabled(true);
    text.requestFocus();
    button.setEnabled(true);
  }

  // Die gewünschte Benutzer-Aktion wird ausgeführt.
  public void actionPerformed(ActionEvent e) {
    Object obj = e.getSource();
    String cmd = e.getActionCommand();

    try {
      if (obj == button) {
        if (cmd.equals("Login")) {
          name = text.getText();
          if (name.length() != 0) {
            login();
          }
        }
        else {
          destroy();
        }
      }

      if (obj == text) {
        if (login) {
          out.write(text.getText());
          out.newLine();
          out.flush();
        }
      }
    }
    catch (IOException ex) {
      area.append(ex.getMessage() + "\n");
      destroy();
    }
    finally {
      text.setText("");
      text.requestFocus();
    }
  }

  // Login beim Server
  private void login() throws IOException {
    socket = new Socket(host, port);
    in = new BufferedReader(
      new InputStreamReader(socket.getInputStream()));
    out = new BufferedWriter(
      new OutputStreamWriter(socket.getOutputStream()));

    out.write(name);
    out.newLine();
    out.flush();
```

```
      login = true;
      label.setText(name);
      button.setText("Logout");

      t = new Thread(this);
      t.start();
    }

    // Logout beim Server
    public void destroy() {
      if (login) {
        try {
          login = false;
          label.setText(" ");
          button.setText("Login");
          t = null;
          if (socket != null)
            socket.close();
          if (in != null)
            in.close();
          if (out != null)
            out.close();
        }
        catch (IOException e) { }
      }
    }

    public void run() {
      try {
        while (Thread.currentThread() == t) {
          final String msg = in.readLine();
          if (msg == null)
            break;

          doUpdate(new Runnable() {
            public void run() {
              area.append(msg + "\n");
            }
          });
        }
      }
      catch (IOException e) { }
    }

    private void doUpdate(Runnable r) {
      try {
        EventQueue.invokeLater(r);
      }
      catch (Exception e) { }
    }
  }
```

***Bild 4.6**: Chat-Client*

4.6 Klassen über das Netz laden

In Java müssen Klassen erst dann physisch vorhanden sein, wenn sie verwendet werden sollen (*dynamisches Laden von Klassen*). Ein so genannter *Klassenlader* ist verantwortlich für das Auffinden und Laden von Klassen.

ClassLoader

Die Klasse `java.lang.ClassLoader` ist eine abstrakte Klasse. Sie enthält die Methode `loadClass`, die die Klasse mit dem angegebenen Namen lädt:

```
public Class<?> loadClass(String name) throws ClassNotFoundException
```

Ein eigener Klassenlader (als Subklasse von `ClassLoader`) kann z.B. eine Klasse über das Netz laden.

Hierzu muss nur die `ClassLoader`-Methode

```
protected Class<?> findClass(String name)
    throws ClassNotFoundException
```

in geeigneter Weise überschrieben werden.

Programm 4.4

Im Folgenden entwickeln wir einen Server (`ClassServer`), der auf Anfrage eine Klasse in seinem Serververzeichnis sucht und den Inhalt als Byte-Array liefert.

Der Client (`NetworkClassLoader`) überschreibt als Subklasse von `ClassLoader` die Methode `findClass`. Dazu nutzt er die `ClassLoader`-Methode `defineClass`, um aus dem vom Server gelieferten Byte-Array ein `Class`-Objekt zu erzeugen:

```
protected final Class<?> defineClass(
    String name, byte[] b, int off, int len)
```

Beim Einsatz eines Programms mit eigenem Klassenlader ist generell zu beachten:

Es wird versucht, weitere in der soeben geladenen Klasse referenzierte Klassen mit dem gleichen Klassenlader zu laden, mit dem auch die erste Klasse geladen wurde.

Befindet sich das aufgerufene Programm (die Start-Klasse) und die zu ladende Klasse im gleichen Klassenpfad, so wird diese Klasse vom Standard-Klassenlader des Systems und nicht vom eigenen Klassenlader geladen.

ClassServer

Der Server wird mit *Portnummer* und *Suchpfad* aufgerufen. Da der Name der angeforderten Klasse auch den Paketnamen enthalten kann, werden Punkte durch Schrägstriche ersetzt und somit die einzelnen Teile des Paketnamens auf Verzeichnisnamen abgebildet.

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

public class ClassServer extends Thread {
  private int port;
  private String searchPath;

  public ClassServer(int port, String searchPath) {
    this.port = port;
    this.searchPath = searchPath;
  }

  public void startServer() {
    try {
      ServerSocket server = new ServerSocket(port);

      InetAddress addr = InetAddress.getLocalHost();
      System.out.println("ClassServer auf " +
        addr.getHostName() + "/" + addr.getHostAddress() +
        ":" + port + " gestartet ...");

      while (true) {
        Socket client = server.accept();
        new ClassThread(client, searchPath).start();
      }
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
```

```
private class ClassThread extends Thread {
  private Socket client;
  private String searchPath;

  public ClassThread(Socket client, String searchPath) {
    this.client = client;
    this.searchPath = searchPath;
  }

  public void run() {
    String clientAddr = client.getInetAddress().getHostAddress();
    int clientPort = client.getPort();
    System.out.println("Verbindung zu " +
      clientAddr + ":" + clientPort + " aufgebaut");

    try {
      BufferedReader in = new BufferedReader(
        new InputStreamReader(client.getInputStream()));
      ObjectOutputStream out = new ObjectOutputStream(
        client.getOutputStream());
      out.flush();

      String name = in.readLine();

      name = name.replace('.', '/');
      String classPath = searchPath + name + ".class";

      File file = new File(classPath);
      int length = (int) file.length();

      FileInputStream fis = new FileInputStream(file);
      byte data[] = new byte[length];
      int cnt = 0;
      while (cnt < length) {
        int result = fis.read(data, cnt, length - cnt);
        if (result == -1)
          break;
        cnt += result;
      }
      fis.close();

      if (data != null) {
        out.writeObject(data);
        out.flush();
      }

      in.close();
      out.close();
      System.out.println("\"" + classPath + "\" gesendet");
    }
    catch (IOException e) {
      System.err.println(e);
    }
```

```
      finally {
        try {
          if (client != null)
            client.close();
        }
        catch (IOException e) { }

        System.out.println("Verbindung zu " +
          clientAddr + ":" + clientPort + " abgebaut");
      }
    }
  }

  public static void main(String[] args) {
    if (args.length != 2) {
      System.err.println("java ClassServer <port> <searchPath>");
      System.exit(1);
    }

    int port = Integer.parseInt(args[0]);
    String searchPath = args[1];

    searchPath = searchPath.replace('\\', '/');
    if (!searchPath.endsWith("/"))
      searchPath += "/";

    new ClassServer(port, searchPath).startServer();
  }
}
```

NetworkClassLoader

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

public class NetworkClassLoader extends ClassLoader {
  private String host;
  private int port;

  public NetworkClassLoader(String host, int port) {
    this.host = host;
    this.port = port;
  }

  public Class<?> findClass(String name) {
    byte[] b = loadClassData(name);
    return defineClass(name, b, 0, b.length);
  }

  private byte[] loadClassData(String name) {
    Socket socket = null;
    try {
      socket = new Socket(host, port);

      ObjectInputStream in =
         new ObjectInputStream(socket.getInputStream());
      PrintWriter out = new PrintWriter(
         socket.getOutputStream(), true);
```

```
      out.println(name);
      Object obj = in.readObject();
      in.close();
      out.close();
      return (byte[])obj;
    }
    catch (ConnectException e) {
      throw new RuntimeException(
        "Verbindung zum Server konnte nicht hergestellt werden.");
    }
    catch (EOFException e) {
      throw new RuntimeException("Klasse wurde nicht gefunden.");
    }
    catch (Exception e) {
      throw new RuntimeException(e);
    }
    finally {
      try {
        if (socket != null)
          socket.close();
      }
      catch (IOException e) { }
    }
  }
}
```

Das Programm `Start` wird mit den folgenden Parametern aufgerufen: *Start*

- Name bzw. IP-Adresse des entfernten Rechners
- Portnummer des entfernten Rechners
- Name der zu ladenden Klasse, die die `main`-Methode enthält
- evtl. Aufrufparameter für die `main`-Methode

Das *Reflection-API* wird genutzt, um die `main`-Methode der geladenen Klasse mit evtl. Parametern aufzurufen.

```
public class Start {
  public static void main(String[] args) {
    if (args.length < 3) {
      System.err.println(
        "java Start <host> <port> <className> [<param1> ...]");
      System.exit(1);
    }

    String host = args[0];
    int port = Integer.parseInt(args[1]);
    String className = args[2];

    try {
      NetworkClassLoader loader = new NetworkClassLoader(host, port);
      Class c = loader.loadClass(className);
```

```
      // Bereitstellung der main-Methode
      java.lang.reflect.Method m = c.getMethod(
        "main", new Class[] {String[].class});

      String[] params = new String[args.length - 3];
      for (int i = 3; i < args.length; i++)
        params[i - 3] = args[i];

      // Aufruf der main-Methode mit evtl. Parametern
      m.invoke(null, new Object[] {params});
    }
    catch (RuntimeException e) {
      System.err.println(e.getMessage());
    }
    catch (Exception e) {
      System.err.println(e);
    }
  }
}
```

Zur Demonstration des Verfahrens kann z. B. das Programm 4.3 (`ChatClient`) genutzt werden.

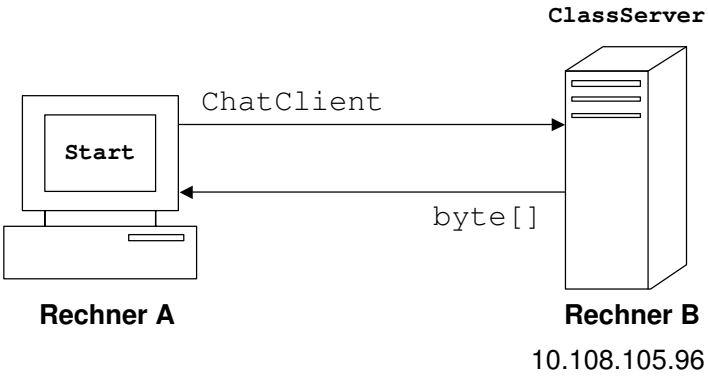


***Bild 4.7**: Dynamisches Laden über das Netz*

Das Laden unbekannter Klassen aus dem Netz stellt ein Sicherheitsrisiko dar. Eine Möglichkeit, sich zu schützen, besteht darin, den *Security Manager* zu nutzen und nur so viele Zugriffsrechte wie nötig für die Ausführung der Anwendung zuzulassen.

Die Zugriffsrechte, passend zum obigen Beispiel, werden in einer *Policy-Datei* beschrieben:

policy.txt

```
grant {
  permission java.lang.RuntimePermission "createClassLoader";
  permission java.lang.RuntimePermission "exitVM";
  permission java.net.SocketPermission "10.108.105.96:40000",
    "connect";
  permission java.net.SocketPermission "10.108.105.96:50000",
    "connect";
  permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
};
```

Aufruf des Start-Programms (Kommando in einer Zeile):

```
java -Djava.security.manager -Djava.security.policy= policy.txt
-cp build Start 10.108.105.96 40000 ChatClient 10.108.105.96 50000
```

4.7 Remote Procedure Call

In diesem Abschnitt entwickeln wir ein allgemeingültiges einfaches Verfahren zum Aufruf von Methoden, die auf einem entfernten Rechner implementiert sind (RPC = Remote Procedure Call).

Programm 4.5

Einzige Voraussetzung ist: Die Parameterwerte einer entfernten Methode müssen Objekte sein. Die Klassen aller Objekte (Rückgabewert und Parameterwerte) müssen das Interface `java.io.Serializable` implementieren.

Der Client ruft die gewünschte Methode mit Hilfe von

```
Object call(String name, Object[] params)
```

der Klasse `RPCClient` auf. `name` ist der Methodenname, `params` enthält die Parameterwerte.

Beispiel:

Lautet die Deklaration der entfernten Methode

```
public int getSumme(Integer x, Integer y),
```

so sieht der Codeabschnitt des Clients für den Aufruf beispielsweise so aus:

```
Object[] params = {10, 33};
int summe = (Integer) rpcClient.call("getSumme", params);
```

RPCClient

Die Klasse `RPCClient` nutzt die Methoden `writeObject` und `readObject` der Klassen `ObjectOutputStream` bzw. `ObjectInputStream`, um den Methodennamen, das Parameter-Array und den Rückgabewert über das Netz zu transferieren. Bei einem Rückgabewert vom Typ `Exception` wird eine Ausnahme dieses Typs ausgelöst.

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

public class RPCClient {
  private String host;
  private int port;

  public RPCClient(String host, int port) {
    this.host = host;
    this.port = port;
  }

  public Object call(String name, Object[] params) throws Exception {
    Socket socket = null;
    try {
      socket = new Socket(host, port);

      ObjectOutputStream out = new ObjectOutputStream(
         socket.getOutputStream());
      out.writeObject(name);
      out.writeObject(params);
      out.flush();

      ObjectInputStream in = new ObjectInputStream(
         socket.getInputStream());
      Object ret = in.readObject();

      in.close();
      out.close();

      if (ret instanceof Exception)
        throw (Exception) ret;

      return ret;
    }
    finally {
      try {
        if (socket != null)
          socket.close();
      }
      catch (IOException e) { }
    }
  }
}
```

RPCServer

Die Klasse `RPCServer` nutzt das *Reflection-API*. Sie lädt diejenige Klasse (hier als *Service* bezeichnet), die die Implementierung der entfernten Methode enthält, und erzeugt eine Instanz dieser Klasse.

Anhand des Methodennamens und der Parameterwerte, die vom Client als Objekte übermittelt wurden, wird mit Hilfe der `Class`-Methode `getMethod` die gewünschte Methode des Service als `Method`-Objekt zur Verfügung gestellt.

Anschließend wird mit `invoke` die Service-Methode aufgerufen. Wenn die Service-Methode eine Ausnahme auslöst, so löst `invoke` die Ausnahme `java.lang.reflect.InvocationTargetException` aus.

Die `InvocationTargetException`-Methode

`Throwable` **`getTargetException`**`()`

gibt die von der aufgerufenen Service-Methode ausgelöste Ausnahme zurück.

Der `RPCServer` liefert Ausnahmen (Typ `Exception`), die beim Suchen bzw. bei der Ausführung der Service-Methode auftreten können, als "normales" Ergebnisobjekt des Aufrufs zurück.

```
import java.io.*;
import java.net.*;
import java.lang.reflect.*;

public class RPCServer extends Thread {
  private int port;
  private String service;

  public RPCServer(int port, String service) {
    this.port = port;
    this.service = service;
  }

  public void startServer() throws Exception {
    ServerSocket server = new ServerSocket(port);

    InetAddress addr = InetAddress.getLocalHost();
    System.out.println("RPCServer auf " +
      addr.getHostName() + "/" + addr.getHostAddress() +
      ":" + port + " gestartet ...");
    System.out.println("Service: " + service);

    Class serviceClass = Class.forName(service);
    Object serviceObject = serviceClass.newInstance();

    while (true) {
      Socket client = server.accept();
      new RPCThread(client, serviceObject).start();
    }
  }

  private class RPCThread extends Thread {
    private Socket client;
    private Object serviceObject;

    public RPCThread(Socket client, Object serviceObject) {
      this.client = client;
      this.serviceObject = serviceObject;
    }
```

```
    public void run() {
      try {
        ObjectInputStream in = new ObjectInputStream(
          client.getInputStream());
        String name = (String) in.readObject();
        Object[] params = (Object[]) in.readObject();

        Class[] types = new Class[params.length];
        for (int i = 0; i < params.length; i++)
          types[i] = params[i].getClass();

        Object ret = null;
        try {
          Method m = serviceObject.getClass().getMethod(name, types);
          ret = m.invoke(serviceObject, params);
        }
        // Eine Ausnahme wird als Ergebnisobjekt zurückgeliefert
        catch (InvocationTargetException e) {
          ret = e.getTargetException();
        }
        catch (Exception e) {
          ret = e;
        }

        ObjectOutputStream out = new ObjectOutputStream(
          client.getOutputStream());
        out.writeObject(ret);
        out.flush();

        in.close();
        out.close();
      }
      catch (Exception e) {
        System.err.println(e);
      }
      finally {
        try {
          if (client != null)
            client.close();
        }
        catch (IOException e) { }
      }
    }

    public static void main(String[] args) throws Exception {
      if (args.length != 2) {
        System.err.println("java RPCServer <port> <service>");
        System.exit(1);
      }

      int port = Integer.parseInt(args[0]);
      String service = args[1];
      new RPCServer(port, service).startServer();
    }
  }
```

Einige beispielhafte Service-Methoden sind in der Klasse `DemoService` implementiert. `TestClient` enthält den Aufruf dieser Methoden.

Test

Demo Service

```
import java.util.*;
import java.io.*;

public class DemoService {
  public String getEcho(String text) {
    return text;
  }

  public int getSumme(Integer x, Integer y) {
    return x + y;
  }

  public Date getDate() {
    return new Date();
  }

  public void sendMessage(String msg) {
    System.out.println(msg);
  }

  public Vector<String> getMessages(String file) throws IOException {
    Vector<String> lines = new Vector<String>();

    BufferedReader in = new BufferedReader(new FileReader(file));
    String line;
    while ((line = in.readLine()) != null) {
      lines.add(line);
    }
    in.close();

    return lines;
  }
}
```

TestClient

```
import java.util.*;

public class TestClient {
  public static void main(String[] args) throws Exception {
    String host = args[0];
    int port = Integer.parseInt(args[1]);

    RPCClient rpcClient = new RPCClient(host, port);

    Object[] params1 = {"Hallo"};
    String s = (String) rpcClient.call("getEcho", params1);
    System.out.println("getEcho: " + s);

    Object[] params2 = {10, 33};
    int summe = (Integer) rpcClient.call("getSumme", params2);
    System.out.println("getSumme: " + summe);
```

```
    Object[] params3 = {};
    Date date = (Date) rpcClient.call("getDate", params3);
    System.out.println("getDate: " + date);

    Object[] params4 = {"Dies ist ein Test."};
    rpcClient.call("sendMessage", params4);

    Object[] params5 = {"msg.txt"};
    Vector<String> v = (Vector<String>) rpcClient.call(
      "getMessages", params5);
    System.out.println("getMessages:");
    for (String msg : v) {
      System.out.println(msg);
    }
  }
}
```

Aufruf des Servers:

```
start java -cp build RPCServer 50000 DemoService
```

Aufruf des Client:

```
java -cp build TestClient localhost 50000
```

4.8 Thread-Pooling

Die Fähigkeit des Servers, mehrere Clients quasi gleichzeitig zu bedienen, wurde bisher so gelöst, dass für jede Anfrage eines Clients ein neuer Thread erzeugt wurde. Im Vergleich zu Prozessen ist die Erzeugung eines Threads weniger aufwändig. Trotzdem sollte man hiermit sparsam umgehen; insbesondere dann, wenn kurzzeitig sehr viele Threads mit kurzer Laufzeit benötigt werden.

Thread-Pool

Ab der Version Java SE 5.0 gibt es die Möglichkeit, einen *Thread-Pool* einzusetzen. Dieser bietet die Möglichkeit, mehrere separate Aufgaben vom selben Thread nacheinander ausführen zu lassen. Nicht mehr benutzte Threads werden in den Pool zurückgelegt und können wiederverwendet werden.

Wir benutzen hier eine spezielle Thread-Pool-Variante, einen so genannten *Cached Thread Pool*. Ein solcher Pool wächst bzw. schrumpft nach Bedarf. Steht eine neue Aufgabe (hier eine Client-Anfrage) zur Bearbeitung an und gibt es keinen "freien" Thread im Pool, so wird ein neuer erzeugt, der die Ausführung übernimmt. Ist die Aufgabe ausgeführt, steht dieser Thread als wieder "freier" Thread im Pool zur Verfügung. Wird er innerhalb von 60 Sekunden nicht benötigt, so wird er terminiert und ist dann nicht mehr verwendbar. Der Pool passt sich also dynamisch den momentanen Anforderungen an und ist optimal für kleinere Aufgaben, die in hoher Zahl kurzfristig anstehen.

Für unsere Zwecke benutzen wir die Klasse `java.util.concurrent.Executors` und das Interface `java.util.concurrent.ExecutorService`.

Die statische `Executors`-Methode

`static ExecutorService` **`newCachedThreadPool`**`()`

erzeugt einen *Cached Thread Pool* und liefert diesen als Objekt vom Typ `ExecutorService` zurück.

Das Interface `ExecutorService` enthält u.a. die folgenden Methoden:

`void` **`execute`**`(Runnable task)`
führt die `run`-Methode von `task` in einem Thread des Pools aus.

`void` **`shutdown`**`()`
bewirkt, dass vor dem Aufruf dieser Methode übergebene Aufgaben noch ausgeführt, neue aber nicht mehr akzeptiert werden; alle vom Pool verwalteten Threads werden dann terminiert.

Damit ein Programm ordnungsgemäß beendet werden kann, muss der Pool mit der Methode `shutdown` kontrolliert terminiert werden.

Programm 4.6 demonstriert den Einsatz eines Thread-Pools für einen Server. ***Programm 4.6***

Server

```
import java.io.*;
import java.net.*;
import java.util.concurrent.*;

public class Server {
  private int port;
  private ExecutorService pool;

  public Server(int port) {
    this.port = port;
  }

  public void startServer() {
    try {
      ServerSocket server = new ServerSocket(port);
      System.out.println("Server gestartet ...");

      pool = Executors.newCachedThreadPool();

      while (true) {
        Socket client = server.accept();
        pool.execute(new Handler(client));
      }
    }
```

```
      catch (IOException e) {
        System.err.println(e);
        pool.shutdown();
      }
    }

    private class Handler implements Runnable {
      private Socket client;

      public Handler(Socket client) {
        this.client = client;
      }

      public void run() {
        try {
          BufferedReader in = new BufferedReader(
            new InputStreamReader(client.getInputStream()));
          PrintWriter out = new PrintWriter(
            client.getOutputStream(), true);

          String input = in.readLine();
          if (input != null) {
            int delay = 5000 + (int) (Math.random() * 5000);
            try {
              Thread.sleep(delay);
            }
            catch (InterruptedException e) { }
            out.println("Auftrag " + input + " nach " + delay +
              " ms abgeschlossen");
          }

          in.close();
          out.close();
        }
        catch (IOException e) {
          System.err.println(e);
        }
        finally {
          try {
            if (client != null)
              client.close();
          }
          catch (IOException e) { }
        }
      }
    }

    public static void main(String[] args) {
      int port = Integer.parseInt(args[0]);
      new Server(port).startServer();
    }
  }
```

4.9 Ein Framework für TCP-Server

Die Beispiele der vorhergehenden Abschnitte zeigen, dass die Implementierungen der TCP-Server im Großen und Ganzen fast immer dem gleichen Muster folgen. Es liegt also nahe, die immer wiederkehrenden Codeteile zu standardisieren und als *Framework* für eigene Server-Implementierungen anzubieten.

Das Framework besteht aus den beiden Klassen `TCPServer` und `AbstractHandler`, die zum Paket `tcpframework` gehören. ***Programm 4.7***

Der Konstruktor von `TCPServer` erwartet eine Portnummer und die Klasse des Handlers, der die eigentliche Kommunikation mit dem Client durchführt. Hier wird das `Class`-Objekt des Handlers angegeben. Der Handler ist eine Subklasse von `AbstractHandler`. Innerhalb des Konstruktors werden ein `ServerSocket`-Objekt und ein Thread-Pool (siehe Kapitel 4.8) erzeugt.

`TCPServer` implementiert das Interface `Runnable`. Innerhalb der `run`-Methode wird in einer Schleife die Methode `accept` aufgerufen. Mit dem zurückgegebenen `Socket`-Objekt wird die `handle`-Methode aufgerufen. Diese erzeugt eine neue Instanz des Handlers und ruft für diese Instanz die Methode `handle` mit den Referenzen für das `Socket`-Objekt und den Thread-Pool auf.

Die Methode `stopServer` schließt das `ServerSocket`-Objekt und terminiert den Thread-Pool.

TCPServer

```
package tcpframework;

import java.io.IOException;
import java.net.Socket;
import java.net.ServerSocket;
import java.net.SocketException;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class TCPServer implements Runnable {
  private int port;
  private Class handlerClass;
  private ServerSocket serverSocket;
  private ExecutorService pool;

  public TCPServer(int port, Class handlerClass) throws IOException {
    this.port = port;
    this.handlerClass = handlerClass;
    serverSocket = new ServerSocket(port);
    pool = Executors.newCachedThreadPool();
  }
```

```
  public void run() {
    try {
      while (true) {
        // Beim Aufruf von stopServer() wird eine SocketException
        // ausgelöst
        Socket clientSocket = serverSocket.accept();
        handle(clientSocket);
      }
    }
    catch (SocketException e) { }
    catch (Exception e) {
      System.err.println(e);
    }
  }

  public void stopServer() {
    try {
      serverSocket.close();
    }
    catch (IOException e) { }
    pool.shutdown();
  }

  private void handle(Socket clientSocket) throws Exception {
    AbstractHandler handler = (AbstractHandler) handlerClass.
      newInstance();
    handler.handle(clientSocket, pool);
  }
}
```

Ein konkreter Handler ist von `AbstractHandler` abgeleitet und implementiert die `run`-Methode. Diese wird durch die `handle`-Methode in einem Thread des Pools ausgeführt. Zudem wird das `Socket`-Objekt mit Hilfe der Methode `getClientSocket` dem Handler zur Verfügung gestellt.

AbstractHandler

```
package tcpframework;

import java.net.Socket;
import java.util.concurrent.ExecutorService;

public abstract class AbstractHandler implements Runnable {
  private Socket clientSocket;

  public Socket getClientSocket() {
    return clientSocket;
  }

  public void handle(Socket clientSocket, ExecutorService pool) {
    this.clientSocket = clientSocket;
    pool.execute(this);
  }

  public abstract void run();
}
```

Das Framework wird nun in einem Beispiel verwendet. Hierbei handelt es sich um den *Echo-Server* aus Kapitel 4.3.

Anwendung des Frameworks

EchoServer

```
import java.io.IOException;
import tcpframework.*;

public class EchoServer {
  public static void main(String[] args) throws IOException {
    int port = Integer.parseInt(args[0]);

    TCPServer server = new TCPServer(port, EchoHandler.class);

    // Server wird in einem Thread gestartet
    Thread t = new Thread(server);
    t.start();
    System.out.println("Server gestartet ...");

    // blockiert, bis RETURN eingegeben wurde
    System.in.read();

    server.stopServer();
    System.out.println("Server gestoppt ...");
  }
}
```

Der Server wird nach Betätigung der Return-Taste gestoppt; allerdings erst dann, wenn alle momentan aktiven Client-Bearbeitungen beendet sind.

EchoHandler

```
import java.io.*;
import java.net.Socket;
import tcpframework.*;

public class EchoHandler extends AbstractHandler {
  public void run() {
    Socket clientSocket = getClientSocket();
    String clientAddr = clientSocket.getInetAddress().
      getHostAddress();

    int clientPort = clientSocket.getPort();
    System.out.println("Verbindung zu " +
      clientAddr + ":" + clientPort + " aufgebaut");

    try {
      BufferedReader in = new BufferedReader(
        new InputStreamReader(clientSocket.getInputStream()));
      PrintWriter out = new PrintWriter(
         clientSocket.getOutputStream(), true);

      out.println("Server ist bereit ...");

      String input;
```

```
      while ((input = in.readLine()) != null) {
        out.println(input);
      }

      in.close();
      out.close();
    }
    catch (IOException e) {
      System.err.println(e);
    }
    finally {
      try {
        if (clientSocket != null)
          clientSocket.close();
      }
      catch (IOException e) { }

      System.out.println("Verbindung zu " +
        clientAddr + ":" + clientPort + " abgebaut");
    }
  }
}
```

Um mit dem Client zu kommunizieren, wird in der `run`-Methode das `Socket`-Objekt über die Methode `getClientSocket` erfragt. Dann erfolgt das Empfangen und Senden der Daten.

4.10 Aufgaben

1. Entwickeln Sie einen TCP-Server, der als Reaktion auf die Verbindungsaufnahme durch den Client an diesen einen in einer Datei gespeicherten Text sendet und dann von sich aus die Verbindung beendet.

   Testen Sie den Server mit Hilfe von Telnet:
   `telnet localhost 50000`

2. Entwickeln Sie einen TCP-Server, der die *aktuelle Systemzeit des Servers* als Zeichenkette liefert. Programmieren Sie auch einen passenden Client hierzu.

3. Entwickeln Sie einen Echo-Server, der die parallele Bedienung mehrerer Clients ermöglicht, aber die Anzahl der gleichzeitig aktiven Threads auf eine vorgegebene Zahl beschränkt.

4. Entwickeln Sie für den File-Server aus Kapitel 4.4 einen Client mit grafischer Oberfläche.

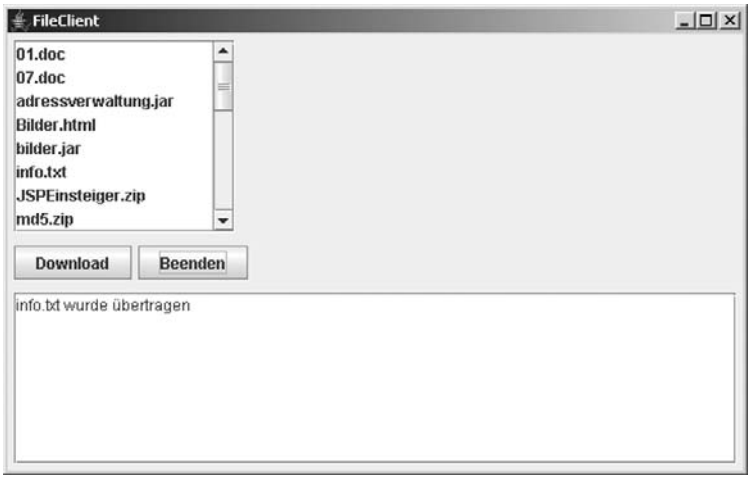


***Bild 4.8**: File-Client*

5. Aus einer Bücher-Datenbank sollen zu einer vorgegebenen Buchnummer Angaben zum Buch (Autor und Titel) über SQL abgefragt werden. Nutzen Sie das RPC-Verfahren aus Kapitel 4.7 und entwickeln Sie hierzu einen Service mit der Methode

   ```
   public Buch getBuch(String id) throws Exception
   ```

   und einen Client, der diese entfernte Methode aufruft.

   Die Klasse `Buch` soll die Angaben zum Buch als Attribute mit den entsprechenden set- und get-Methoden enthalten.

5 Implementierung eines HTTP-Servers

Ziel dieses Kapitels ist es, eine Einführung in das *Hypertext Transfer Protocol* (HTTP) zu geben und einen einfachen Webserver zu entwickeln, der in einer erweiterten Fassung dynamische Webseiten erzeugen kann.

5.1 Das Protokoll HTTP

HTTP ist ein Protokoll der Anwendungsschicht im TCP/IP-Schichtenmodell und regelt insbesondere, wie ein Webbrowser mit einem Webserver im World Wide Web (WWW) kommuniziert. HTTP verwendet auf der Transportschicht TCP.

Damit ein Webbrowser eine Webseite im WWW abrufen kann, muss er sie zunächst adressieren.

Ein *Uniform Resource Locator* (URL) ist eine standardisierte Adresse, mit der eine beliebige Ressource (z.B. eine HTML-Seite, ein GIF-Bild, eine PDF-Datei, ein Programm) lokalisiert werden kann. *URL*

So lokalisiert z.B.

```
http://www.hs-niederrhein.de/index.html
```

die Website der Hochschule Niederrhein.

Der URL hat im Allgemeinen den folgenden Aufbau:

```
protocol://host[:port][/path][/file][#section]
```

Hierbei sind die eingeklammerten Teile optional.

Die Angaben bedeuten:

`protocol`	Protokollname, hier: `http`
`host`	Domain-Name oder IP-Adresse des Rechners
`port`	Portnummer, unter der der Server läuft; die standardmäßige Portnummer für einen HTTP-Server ist 80 und muss nicht angegeben werden
`path`	Name eines Verzeichnisses auf dem Server; die Angabe ist relativ zur Wurzel des Webverzeichnisses
`file`	Name einer Datei in dem spezifizierten Verzeichnis
`section`	verweist auf eine bestimmte Stelle innerhalb einer HTML-Seite

Ablauf einer HTTP-Transaktion

Die Interaktion zwischen HTTP-Client und HTTP-Server für eine Anfrage umfasst die folgenden Schritte:

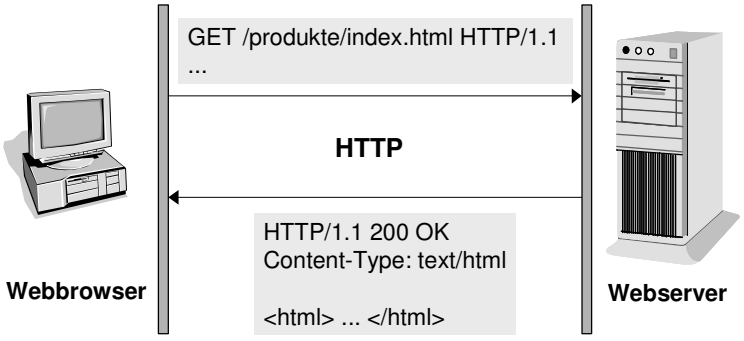
1. Der Server wartet auf eine eingehende HTTP-Anfrage.
2. Der Client erzeugt einen URL `http://...`
3. Der Client versucht, eine TCP-Verbindung zum Server aufzubauen.
4. Der Server akzeptiert den Verbindungswunsch.
5. Der Client sendet eine Nachricht (HTTP-Anfrage) an den Server und fordert die Ressource mit dem spezifizierten URL an.
6. Der Server verarbeitet die Anfrage (z.B. Ausführung einer Datenbankabfrage und Generierung einer HTML-Seite mit dem Abfrageergebnis).
7. Der Server sendet eine Rückantwort (HTTP-Antwort) an den Client, die die angeforderte Ressource oder eine Fehlermeldung enthält.
8. Der Client verarbeitet die Antwort.
9. Der Client und/oder der Server schließen die TCP-Verbindung.

HTTP ist zustandslos

Der Server hat keine Kenntnis über vorangegangene Anfragen desselben Client. Jede Anfrage wird unabhängig von vorhergehenden Anfragen bearbeitet. HTTP ist also ein *zustandsloses Protokoll*.

Für jede HTTP-Anfrage wird bei HTTP 1.0 in der Regel eine neue TCP-Verbindung aufgebaut. Enthält z.B. eine angeforderte HTML-Seite mehrere Grafiken, so muss jede dieser Grafiken separat angefordert werden (jeweils mit Verbindungsaufbau und -abbau).

***Bild 5.1**: Eine HTTP-Transaktion*



HTTP 1.0 ist im RFC 1945 der IETF spezifiziert (siehe `http://www.ietf.org/rfc/rfc1945.txt`). *HTTP 1.0*

Die Version *HTTP 1.1* unterstützt so genannte *persistente Verbindungen*. Während die TCP-Verbindung steht, können *mehrere* HTTP-Anfragen über diese Verbindung durchgeführt werden. Die Verbindung kann vom Client oder Server abgebaut werden. *HTTP 1.1*

HTTP 1.1 ist im RFC 2616 der IETF spezifiziert (siehe `http://www.ietf.org/rfc/rfc2616.txt`).

Das folgende Programm zeigt den Inhalt der Nachricht (HTTP-Anfrage), die ein Webbrowser an den Webserver schickt.

Programm 5.1

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

public class Reporter {
  private int port;
  private String file;

  public Reporter(int port, String file) {
    this.port = port;
    this.file = file;
  }

  public void doWork() {
    ServerSocket server = null;
    Socket client = null;
    InputStream in = null;
    OutputStream out = null;

    try {
      server = new ServerSocket(port);
      client = server.accept();

      in = client.getInputStream();
      out = new FileOutputStream(file);

      // Timeout, da die Leseschleife nicht beendet wird.
      client.setSoTimeout(3000);

      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    }
    catch (IOException e) {
      System.err.println(e);
    }
    finally {
      try {
        if (in != null)
          in.close();
```

```
        if (out != null) {
          out.flush();
          out.close();
        }
        if (client != null)
          client.close();
        if (server != null)
          server.close();
      }
      catch (IOException e) { }
    }
  }

  public static void main(String[] args) {
    if (args.length != 2) {
      System.err.println("java Reporter <port> <file>");
      System.exit(1);
    }

    int port = Integer.parseInt(args[0]);
    String file = args[1];
    new Reporter(port, file).doWork();
  }
}
```

Das Programm protokolliert die empfangenen Daten in einer Datei, deren Name beim Aufruf als Parameter mitgegeben wird, und beendet sich nach drei Sekunden selbst.

Test

Aufruf des Programms:

```
java -cp build Reporter 50000 log.txt
```

Anforderung einer fiktiven HTML-Seite im Webbrowser (hier: Firefox):

```
http://localhost:50000/abc/xyz.html
```

Inhalt der Datei *log.txt*:

```
GET /abc/xyz.html HTTP/1.1
Host: localhost:50000
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; de; …
Accept: text/xml,application/xml,application/xhtml+xml,text/html;…
Accept-Language: de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

HTTP-Anfrage

Eine *HTTP-Anfrage* hat den folgenden Aufbau:

- *Kopfzeile*
  Sie enthält die *HTTP-Methode* (im Beispiel `GET`), den Namen der angeforderten *Ressource* ohne Protokoll und Domain-Namen (im Beispiel `/abc/xyz.html`) und die verwendete *Protokollversion* (im Beispiel HTTP/1.1).
- *Anfrageparameter (optional)*
  Anfrageparameter liefern dem Server zusätzliche Informationen über den Client und seine Anfrage. Jeder Parameter benutzt eine eigene Zeile und besteht aus dem Namen, einem Doppelpunkt und dem Wert.
- eine *Leerzeile*
- *Nutzdatenteil (optional)*
  Hier stehen z.B. die in einem Formular eingetragenen Daten bei Anwendung der HTTP-Methode `POST`.

Kopfzeile, Anfrageparameter und Leerzeile enden jeweils mit *Carriage Return* und *Linefeed*: `\r\n`.

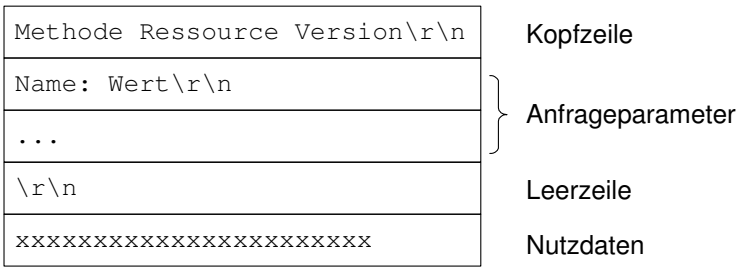


***Bild 5.2**: Struktur einer HTTP-Anfrage*

Im einfachsten Fall reicht eine Anfrage der Form

```
GET /index.html HTTP/1.0
```

aus, um bereits von einem Webserver verstanden zu werden.

HTTP-Methoden

Die *HTTP-Methode* spezifiziert die vom Server durchzuführende Aktion.

Wichtige HTTP-Methoden sind:

- `GET`
  Diese Methode fordert eine Ressource an.
- `POST`
  Diese Methode überträgt Benutzerdaten an den Server.
- `HEAD`
  Diese Methode fordert Informationen über die Ressource an; die Ressource selbst wird nicht benötigt.

- `PUT`
  Diese Methode wird verwendet, um eine Ressource auf dem Server abzulegen.
- `DELETE`
  Diese Methode wird verwendet, um eine Ressource auf dem Server zu löschen.

`PUT` und `DELETE` werden aus Sicherheitsgründen von den meisten Webservern ignoriert.

Anfrageparameter

Anfrageparameter können in beliebiger Reihenfolge angegeben werden. Groß- und Kleinschreibung wird ignoriert.

Die gebräuchlichsten Anfrageparameter sind (siehe obiges Testbeispiel):

- `Host`
  Rechnername und optionale Portnummer des Servers
- `User-Agent`
  Kenndaten über den HTTP-Client
- `Connection`
  Dieser Parameter wird benutzt, um eine persistente TCP-Verbindung anzufordern bzw. zu schließen.
- `Content-Length`
  Länge der Daten (in Byte) im Nutzdatenteil
- `Accept-Language`
  Dieser Parameter gibt die vom Client bevorzugte Sprache an. Der Server kann dann z.B. eine HTML-Seite, die in mehreren Sprachvarianten vorliegt, in der vom Client gewünschten Sprache senden.
- `Accept-Encoding`
  Mit diesem Parameter gibt der Client an, welche Komprimierungsalgorithmen er versteht. Der Server kann z.B. große Dateien komprimieren, um die Übertragungszeit zu minimieren.
- `Accept-Charset`
  Dieser Parameter gibt die vom Client bevorzugten Zeichensätze an.
- `Accept`
  Dieser Parameter gibt die vom Client akzeptierten Medientypen an. Die gültigen Parameterwerte sind durch den MIME-Standard definiert.

MIME

MIME steht für den Standard *Multipurpose Internet Mail Extension*, der ursprünglich für E-Mails entworfen wurde.

MIME-Formatangaben werden von HTTP-Clients und HTTP-Servern benutzt. Clients nutzen sie, um dem Server mitzuteilen,

welche Medientypen sie handhaben können. Server nutzen sie, um den Client über den Inhaltstyp der gesendeten Ressource zu informieren.

Die MIME-Formatangabe besteht aus einer Typ- und einer Subtypangabe:

```
typ/subtyp
```

Der MIME-Standard ist im RFC 1521 der IETF spezifiziert (siehe `http://www.ietf.org/rfc/rfc1521.txt`).

Beispiele von Medientypen

Typ/Subtyp	Beschreibung und übliche Erweiterung
`text/html`	HTML-Datei (*.htm, *.html)
`text/plain`	ASCII-Text (*.txt)
`text/xml`	XML-Datei (*.xml, *.dtd)
`image/gif`	GIF-Bild (*.gif)
`image/jpeg`	JPEG-Bild (*.jpeg, *.jpg)
`image/png`	PNG-Bild (*.png)
`application/pdf`	PDF-Datei (*.pdf)
`application/octet-stream`	Binärdaten (*.bin, *.exe)
`application/zip`	ZIP-Datei (*.zip)

Für nicht standardisierte Subtypen wird das Präfix `x-` benutzt, z.B. bezeichnet `audio/x-wav` Audio-Dateien *.wav.

Daten zum Server senden

Mit der HTTP-Methode `POST` können Benutzerdaten zum Server geschickt werden.

Das folgende Beispiel zeigt ein HTML-Formular, dessen Daten durch Betätigen des Buttons "Senden" zum Server (hier Programm 5.1) gesendet werden.

HTML-Code des Formulars

```
<html>
<head><title>POST</title></head>
<body bgcolor="lightgrey">
<form action="http://localhost:50000/xxx" method="POST">
<pre>
Artikelnummer: <input type="text" name="nr" size="5"/>
Bezeichnung:   <input type="text" name="bez" size="30"/>
Preis:         <input type="text" name="preis" size="10"/>
</pre>
Beschreibung:<br/>
<textarea name="beschr" cols="60" rows="5"></textarea>
<p/>
```

```
<input type="submit" value="Senden"/>
<input type="reset" value="Zurücksetzen"/>
</form>
</body>
</html>
```

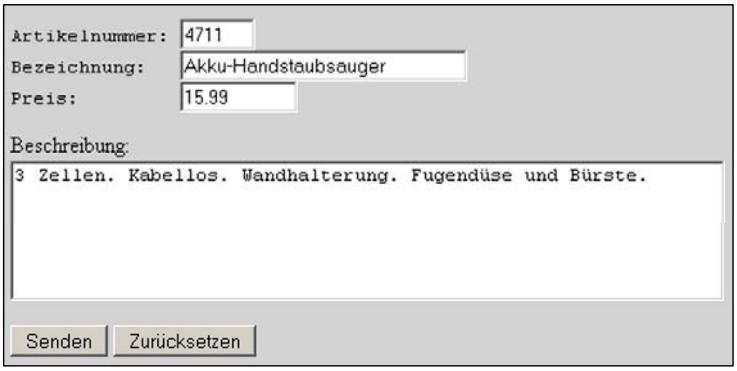


***Bild 5.3**: Ein Formular*

Das Programm *Reporter* protokolliert:

```
POST /xxx HTTP/1.1
Host: localhost:50000
…
Content-Length: 112

nr=4711&bez=Akku-Handstaubsauger&preis=15.99&beschr=3+Zellen.+
Kabellos.+Wandhalterung.+Fugend%FCse+und+B%FCrste.
```

URL-codiertes Format

Die im Formular eingetragenen Daten werden im *URL-codierten Format* (`application/x-www-form-urlencoded`) übertragen.

Die im Formular definierten Variablennamen (im Beispiel: *nr*, *bez*, *preis*, *beschr*) sind mit den vom Benutzer eingegebenen Werten verknüpft. Variable und Wert werden jeweils durch ein Gleichheitszeichen voneinander getrennt. Die einzelnen Variable/Wert-Paare sind durch das Zeichen `&` getrennt. Bestimmte Sonderzeichen, die eine spezielle Bedeutung haben (z.B. `=` und `&`), werden ersetzt durch ein Prozentzeichen, gefolgt vom ASCII-Wert des Zeichens als Hexadezimalwert. Leerzeichen werden durch `+` ersetzt.

Formulare können auch die `GET`-Methode nutzen, um Daten zu übertragen. Die URL-codierten Daten werden nach einem Fragezeichen `?` an den URL angehängt. Im HTML-Code des obigen Formulars muss nur `POST` durch `GET` ersetzt werden. Die Kopfzeile der HTTP-Anfrage hat dann das folgende Aussehen:

```
GET /xxx?nr=4711&bez=Akku-Handstaubsauger&preis=15.99&beschr=
3+Zellen.+Kabellos.+Wandhalterung.+Fugend%FCse+und+B%FCrste. HTTP/1.1
```

Die so codierte Anfragezeichenkette nach dem Fragezeichen wird auch als *Query String* bezeichnet.

Query String

Eine *HTTP-Antwort* hat den folgenden Aufbau:

HTTP-Antwort

- *Kopfzeile*
  Sie besteht aus der *Protokollversion*, dem *Status-Code* und einer optionalen *Status-Meldung*.
- *Antwortparameter (optional)*
  Antwortparameter liefern dem Client zusätzliche Informationen über den Server und die Antwort.
- eine *Leerzeile*
- *Nutzdatenteil (optional)*
  Dieser enthält die angeforderte Ressource.

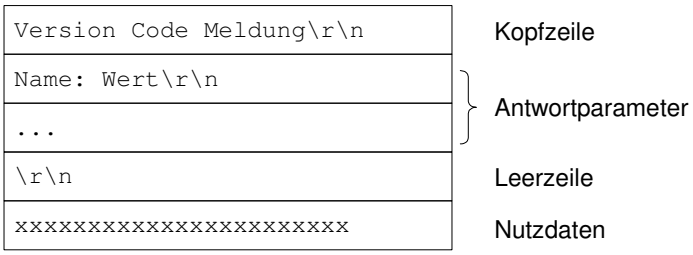


***Bild 5.4**: Struktur einer HTTP-Antwort*

Beispiel:

```
HTTP/1.1 200 OK
Date: Mon, 09 Aug 2005 08:20:30 GMT
Server: Apache/1.3.31 (Win32)
Content-Type: text/html
Content-Length: 249

<html>...</html>
```

Der *Status-Code* teilt dem Client mit, wie die gewünschte Aktion vom Server ausgeführt wurde.

Status-Code

Die Status-Codes sind wie folgt gruppiert:

100 – 199	Informative Meldungen
200 – 299	Die Anfrage war erfolgreich
300 – 399	Die Anfrage wurde weitergeleitet
400 – 499	Die Anfrage war fehlerhaft
500 – 599	Server-Fehler

Codes, die in den folgenden Programmbeispielen genutzt werden, sind:

200	OK
400	Bad Request
404	Not Found
500	Internal Server Error
501	Not Implemented

Einige Antwortparameter

Gebräuchliche *Antwortparameter* sind:

- `Date`
  Aktuelles Datum des Servers zum Zeitpunkt der Beantwortung der Anfrage
- `Server`
  Kenndaten über den HTTP-Server
- `Content-Type`
  MIME-Format der Nutzdaten
- `Content-Length`
  Länge der Daten (in Byte) im Nutzdatenteil. Werden Webseiten dynamisch erzeugt, ist die Länge oft nicht bekannt, weshalb dieser Parameter dann weggelassen wird.

5.2 Ein einfacher File-Server

Programm 5.2

Das folgende Programm ist ein HTTP-Server, der als einzigen Dienst nach Verbindungsaufnahme mit einem Client immer dieselbe Datei sendet. Zu diesem Zweck muss er also die HTTP-Anfrage des Client gar nicht auswerten.

Der Server wird mit dem Dateinamen als Parameter aufgerufen. Anhand der Dateiendung wird der passende MIME-Typ festgelegt. Die HTTP-Antwort besteht aus der Kopfzeile mit Status-Code 200, den beiden Antwortparametern `Content-Type` und `Content-Length`, einer Leerzeile und dem Inhalt der Datei.

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

public class SingleFileServer {
  private int port;
  private File file;
  private String type;

  public SingleFileServer(int port, File file) {
    this.port = port;
    this.file = file;
```

```
    // MIME-Typ festlegen
    String filename = file.getName();
    if (filename.endsWith(".html") || filename.endsWith(".htm"))
      type = "text/html";
    else if (filename.endsWith(".txt") || filename.endsWith(".java"))
      type = "text/plain";
    else if (filename.endsWith(".gif"))
      type = "image/gif";
    else if (filename.endsWith(".jpg"))
      type = "image/jpeg";
    else if (filename.endsWith(".png"))
      type = "image/png";
    else if (filename.endsWith(".pdf"))
      type = "application/pdf";
    else
      type = "application/octet-stream";
  }

  public void startServer() {
    try {
      ServerSocket server = new ServerSocket(port);

      InetAddress addr = InetAddress.getLocalHost();
      System.out.println("SingleFileServer auf " +
        addr.getHostName() + "/" + addr.getHostAddress() +
        ":" + port + " gestartet ...");

      while (true) {
        Socket client = server.accept();
        new SingleFileThread(client).start();
      }
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }

  private class SingleFileThread extends Thread {
    private Socket client;

    public SingleFileThread(Socket client) {
      this.client = client;
    }

    public void run() {
      String clientAddr = client.getInetAddress().getHostAddress();
      int clientPort = client.getPort();
      System.out.println("Verbindung zu " +
        clientAddr + ":" + clientPort + " aufgebaut");

      FileInputStream is = null;
      try {
        BufferedReader in = new BufferedReader(
          new InputStreamReader(client.getInputStream()));
        String line;
        while ((line = in.readLine()) != null && !line.equals("")) { }
```

```
        is = new FileInputStream(file);
        BufferedOutputStream out = new BufferedOutputStream(
          client.getOutputStream());

        String header = "HTTP/1.0 200 OK\r\n" +
          "Content-Type: " + type + "\r\n" +
          "Content-Length: " + file.length() + "\r\n\r\n";

        out.write(header.getBytes());

        int c;
        while ((c = is.read()) != -1) {
          out.write(c);
        }

        out.flush();
        out.close();
        in.close();
      }
      catch (IOException e) {
        System.err.println(e);
      }
      finally {
        try {
          if (client != null)
            client.close();
          if (is != null)
            is.close();
        }
        catch (IOException e) { }

        System.out.println("Verbindung zu " +
          clientAddr + ":" + clientPort + " abgebaut");
      }
    }
  }

  public static void main(String[] args) {
    if (args.length != 2) {
      System.err.println("java SingleFileServer <port> <file>");
      System.exit(1);
    }

    int port = Integer.parseInt(args[0]);
    File file = new File(args[1]);
    if (!file.isFile()) {
      System.err.println(
         "Datei '" + args[1] + "' ist nicht vorhanden");
      System.exit(1);
    }

    new SingleFileServer(port, file).startServer();
  }
}
```

Aufruf des Servers: *Test*

```
java -cp build SingleFileServer 50000 src/SingleFileServer.java
```

Eingabe im Webbrowser:

```
http://localhost:50000/
```

5.3 Ein HTTP-Server für SQL-Abfragen

Wir stellen zunächst Methoden vor, die eine Zeichenkette in ein URL-codiertes Format transformieren bzw. eine so codierte Zeichenkette wieder decodieren können.

Die Klasse `java.net.URLEncoder` enthält eine Klassenmethode zur URL-Codierung einer Zeichenkette. *URLEncoder*

```
static String encode(String s, String enc)
    throws UnsupportedEncodingException
```

wandelt Zeichen aus `s` in das URL-codierte Format "`x-www-form-urlencoded`" um, wobei für die Zeichenkodierung das Codierungsschema `enc` zugrunde gelegt wird (z.B. `ISO-8859-1`).

Wird das Codierungsschema `enc` nicht unterstützt, so löst diese Methode eine Ausnahme vom Typ `java.io.UnsupportedEncodingException` (Subklasse von `IOException`) aus.

Die Klasse `java.net.URLDecoder` dient der Decodierung einer URL-codierten Zeichenkette. *URLDecoder*

```
static String decode(String s, String enc)
    throws UnsupportedEncodingException
```

decodiert eine URL-codierte Zeichenkette, wobei für die Zeichenkodierung das Codierungsschema `enc` zu Grunde gelegt wird.

Das zu entwickelnde Programm `SqlServer` ermöglicht es, beliebige SQL-Anfragen (Abfragen und Änderungen) an eine Datenbank über den Webbrowser zu schicken. ***Programm 5.3***

Dieser spezielle HTTP-Server

- sendet ein Eingabeformular zur Erfassung des SQL-Befehls an den Webbrowser,
- analysiert den SQL-Befehl und führt ihn über JDBC aus,
- sendet evtl. SQL-Fehlermeldungen und
- sendet das Abfrageergebnis bzw. Informationen über Datenbankänderungen an den Webbrowser.

SqlServer

```
import java.io.*;
import java.net.*;
import java.sql.*;
import java.util.*;

public class SqlServer {
  private static final int MAX_ROWS = 1000;
  private static final int MAX_LENGTH = 1000;
  private int port;
  private Properties prop;

  public SqlServer(int port) {
    this.port = port;
  }

  public void startServer() {
    try {
      // Properties einlesen
      prop = new Properties();
      FileInputStream in = new FileInputStream(
         "dbconnect.properties");
      prop.load(in);
      in.close();

      // JDBC-Treiber laden
      Class.forName(prop.getProperty("driver"));

      ServerSocket server = new ServerSocket(port);
      InetAddress addr = InetAddress.getLocalHost();
      System.out.println("SqlServer auf " +
        addr.getHostName() + "/" + addr.getHostAddress() +
        ":" + port + " gestartet ...");

      while (true) {
        Socket client = server.accept();
        new SqlThread(client, prop).start();
      }
    }
    catch (Exception e) {
      System.err.println(e);
      System.exit(1);
    }
  }

  private class SqlThread extends Thread {
    private Socket client;
    private Properties prop;
    private Connection con;
    private BufferedReader in;
    private PrintWriter out;

    public SqlThread(Socket client, Properties prop) {
      this.client = client;
      this.prop = prop;
    }
```

```
  public void run() {
    String clientAddr = client.getInetAddress().getHostAddress();
    int clientPort = client.getPort();
    System.out.println("Verbindung zu " +
      clientAddr + ":" + clientPort + " aufgebaut");

    try {
      // Verbindung zur Datenbank herstellen
      con = DriverManager.getConnection(prop.getProperty("url"),
      prop.getProperty("user"), prop.getProperty("password"));

      in = new BufferedReader(new InputStreamReader(
        client.getInputStream()));
      out = new PrintWriter(client.getOutputStream(), true);

      // HTTP-Request analysieren
      String sql = readRequest();

      // HTML-Formular senden
      sendForm(sql);

      // SQL-Befehl ausführen
      if (sql != null)
        execute(sql);

      in.close();
      out.close();
    }
    catch (Exception e) {
      System.err.println(e);
    }
    finally {
      try {
        if (con != null)
          con.close();
        if (client != null)
          client.close();
      }
      catch (Exception e) { }

      System.out.println("Verbindung zu " +
        clientAddr + ":" + clientPort + " abgebaut");
    }
  }

  // Der Quellcode der folgenden Methoden ist weiter unten
  // abgedruckt:
  private String readRequest() throws IOException { ... }
  private void sendForm(String sql) throws IOException { ... }
  private void execute(String sql) throws SQLException { ... }

}
```

```
  public static void main(String[] args) {
    if (args.length != 1) {
      System.err.println("java SqlServer <port>");
      System.exit(1);
    }

    int port = Integer.parseInt(args[0]);
    new SqlServer(port).startServer();
  }
}
```

Datenbankspezifische Angaben (JDBC-Treiber, URL der Datenbank, User und Passwort) sind in der Datei *dbconnect.properties* zusammengefasst und werden zu Beginn als *Properties* eingelesen.

Innerhalb der `run`-Methode der inneren Klasse `SqlThread` erfolgt die Interaktion mit dem Client.

Zunächst wird die Verbindung zur Datenbank hergestellt. Die Methode `readRequest` analysiert dann die HTTP-Anfrage und liefert den vom Benutzer eingegebenen SQL-Befehl.

Die Rückantwort des Servers enthält eine HTML-Seite, die im oberen Teil ein Formular zur Eingabe eines SQL-Befehls und im unteren Teil das Ergebnis der letzten Anfrage enthält.

Die Methode `sendForm` sendet das Formular mit dem zuletzt eingegebenen SQL-Befehl an den Client zurück.

Die Methode `execute` führt den SQL-Befehl aus und sendet das Ergebnis bzw. eine Fehlermeldung. Damit ist dann die HTML-Seite vollständig erzeugt.

Zu Beginn der Sitzung wird nur ein leeres Formular an den Client geschickt.

Die Methode readRequest

```
private String readRequest() throws IOException {
  String line = in.readLine();
  if (line == null)
    throw new IOException("Kein Input");

  // SQL-Befehle werden über ein Formular mittels POST gesendet
  if (!line.startsWith("POST"))
    return null;

  // Bis zur Leerzeile lesen und Content-Length ermitteln
  int length = 0;
  while ((line = in.readLine()) != null && !line.equals("")) {
    line = line.toLowerCase();
    if (line.startsWith("content-length"))
      length = Integer.parseInt(line.substring(16));
  }
```

```
  // Zeichen nach der Leerzeile einlesen
  int c;
  StringBuilder sb = new StringBuilder();
  for (int i = 0; i < length; i++) {
    c = in.read();
    if (c == -1)
      break;
    sb.append((char) c);
  }

  // Decodierung der "x-www-form-urlencoded" Zeichen
  String query = sb.toString();
  int i = query.indexOf('=');
  if (i < 0)
    return null;
  return URLDecoder.decode(query.substring(i + 1), "ISO-8859-1");
}
```

Es wird nur die HTTP-Methode `POST` zugelassen. Somit muss der Anfrageparameter `Content-Length` vorhanden sein. Die Länge wird extrahiert und in eine Zahl vom Typ `int` konvertiert. Zeichenweise werden nun die Daten des Nutzdatenteils in einen Puffer übertragen. Die Daten nach dem Gleichheitszeichen = werden extrahiert und, da sie URL-codiert sind, decodiert.

Die Methode sendForm

```
private void sendForm(String sql) throws IOException {
  out.print("HTTP/1.0 200 OK\r\n" +
    "Content-Type: text/html\r\n\r\n");
  out.print("<html><head><title>SQL</title></head>");
  out.print("<body bgcolor='lightgrey'>");
  out.print("<b>" + prop.getProperty("url") + "</b><p/>");
  out.print("<form method='POST'>");
  out.print("<textarea cols='80' rows='4' name='sql'>");

  if (sql != null)
    out.print(sql);

  out.print("</textarea><p/>");
  out.print("<input type='submit' value='Senden'><p/>");
  out.print("</form>");

  if (sql == null)
    out.print("</body></html>");

  out.flush();
}
```

Ist der Parameter `sql` gleich `null` (d.h. es wurden keine Benutzerdaten gesendet), so erzeugt diese Methode die komplette HTTP-Antwort: ein HTML-Formular. Andernfalls generiert sie nur den ersten Teil der Antwort: ein Formular mit dem zuletzt gesendeten SQL-Befehl.

Die Methode execute

```
private void execute(String sql) throws SQLException {
  try {
    Statement stmt = con.createStatement();

    if (!stmt.execute(sql)) {
      out.println(stmt.getUpdateCount() + " Zeile(n)");
      stmt.close();
      return;
    }

    // Ausgabe der Zeilen in Form einer HTML-Tabelle

    ResultSet rs = stmt.getResultSet();
    ResultSetMetaData rm = rs.getMetaData();
    int n = rm.getColumnCount();
    String[] align = new String[n];

    out.println("<table bgcolor='white' " +
      "border='1' cellpadding='5'>");

    // Spaltenüberschriften der Tabelle
    out.println("<tr>");
    for (int i = 1; i <= n; i++) {
      // Zahlen werden rechtsbündig ausgerichtet
      if (rm.getColumnType(i) == Types.TINYINT ||
        rm.getColumnType(i) == Types.SMALLINT ||
        rm.getColumnType(i) == Types.INTEGER ||
        rm.getColumnType(i) == Types.BIGINT ||
        rm.getColumnType(i) == Types.REAL ||
        rm.getColumnType(i) == Types.FLOAT ||
        rm.getColumnType(i) == Types.DOUBLE ||
        rm.getColumnType(i) == Types.NUMERIC ||
        rm.getColumnType(i) == Types.DECIMAL)

        align[i-1] = "right";
      else
        align[i-1] = "left";

      out.println("<th align='" + align[i-1] + "'>" +
        rm.getColumnName(i) + "</th>");
    }
    out.println("</tr>");

    // Tabellenzeilen
    // Anzahl Zeilen und Länge eines Spaltenwerts sind begrenzt
    int count = 0;
    while (rs.next()) {
      if (++count > MAX_ROWS)
        break;

      out.println("<tr>");
      for (int i = 1; i <= n; i++) {
        String s = rs.getString(i);
        if (rs.wasNull()) {
          s = "[NULL]";
        }
```

```
        else {
          if (s.length() > MAX_LENGTH) {
            s = s.substring(0, MAX_LENGTH) + " ...";
          }
        }
        out.println("<td valign='top' align='" +
          align[i-1] + "'>" + s + "</td>");
      }
      out.println("</tr>");
    }

    out.println("</table>");
    if (count > MAX_ROWS)
      out.println("Es werden maximal " + MAX_ROWS +
         " Zeilen angezeigt.");

    rs.close();
    stmt.close();
  }
  catch (SQLException e) {
    out.println(e);
  }
  finally {
    out.println("</body></html>");
  }
}
```

Der SQL-Befehl wird ausgeführt (Details zur Handhabung des JDBC-API findet man im Kapitel 2). Das Ergebnis der Datenbankänderung bzw. Abfrage ergibt den zweiten Teil der Antwort an den Client. Das Abfrageergebnis wird in einer HTML-Tabelle dargestellt. Um die an den Browser zu übertragende Datenmenge zu begrenzen, werden maximal `MAX_ROWS` Zeilen und je Spaltenwert einer Zeile maximal `MAX_LENGTH` Zeichen gesendet.

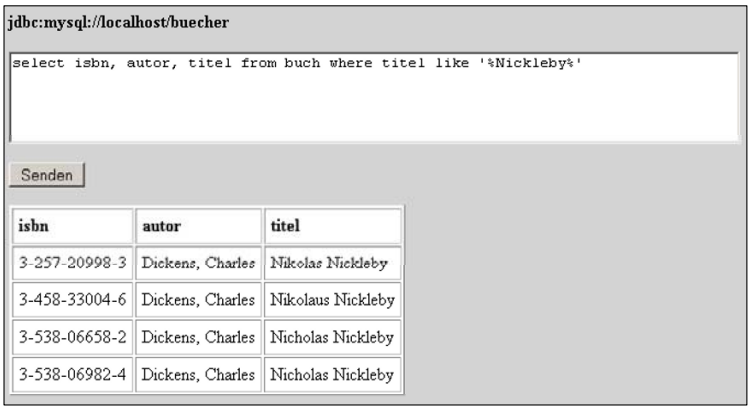


***Bild 5.5**: SQL-Abfrage*

5.4 Ein einfacher Webserver

Programm 5.4

Im Folgenden entwickeln wir einen einfachen Webserver, der nur die HTTP-Methode `GET` versteht und keine Query Strings akzeptiert. Das Programm wird mit zwei Parametern aufgerufen:

- Portnummer *port* und
- Name des so genannten Root-Verzeichnisses *dir*.

dir ist die Wurzel des Webverzeichnisses, das alle abrufbaren Ressourcen enthält.

alias.properties

Der Server unterstützt auch so genannte *virtuelle Verzeichnisse*. In der Datei *alias.properties* sind zu diesem Zweck Zuordnungen in folgender Form festgelegt:

```
logischer Name = physischer Name
```

Beispiele:

Ist das Wurzelverzeichnis `.\web`, so referenziert der URL `http://localhost:50000/index.html` die Datei `.\web\index.html` auf dem Server.

Die Datei *alias.properties* enthält die Zeile:

```
demo = ./web2
```

Der URL `http://localhost:50000/demo/index.html` referenziert die Datei `.\web2\index.html` auf dem Server.

Um zu verhindern, dass Ressourcen, die außerhalb der vorgegebenen Verzeichnisse liegen, vom Browser abgerufen werden, z.B. durch `http://localhost:50000/../abc.txt`, wird in der `main`-Methode ein *Security Manager* installiert:

```
System.setSecurityManager(new SecurityManager());
```

java.policy

Die nötigen Zugriffsrechte werden in der Datei *java.policy* festgelegt:

```
grant {
  permission java.net.SocketPermission "*:1024-","connect,accept";
  permission java.lang.RuntimePermission "shutdownHooks";
  permission java.io.FilePermission "alias.properties", "read";

  // Nur Dateien aus dem hier angegebenen Verzeichnis
  // (inkl. Unterverzeichnissen) können gelesen werden.

  permission java.io.FilePermission "web\\-", "read";
  permission java.io.FilePermission "web2\\-", "read";
};
```

Der Webserver protokolliert jeweils die erste Zeile einer HTTP-Anfrage und kann durch Eingabe der Tastenfolge `Strg+C` beendet werden.

Beispiel (Aufruf in einer Zeile):

```
java -Djava.security.policy=java.policy -cp build
webserver.MiniWebServer 50000 web
```

Ausgabe:

```
MiniWebServer auf pc0806/10.108.105.96:50000 gestartet ...
Wed Aug 30 10:14:19 CEST 2006: 10.108.105.95:3030 GET /
Wed Aug 30 10:14:19 CEST 2006: 10.108.105.95:3031 GET /java.gif
...
Wed Aug 30 10:14:22 CEST 2006: MiniWebServer gestoppt
```

Bild 5.6 zeigt die in diesem Programm verwendeten Klassen und ihre Abhängigkeiten.

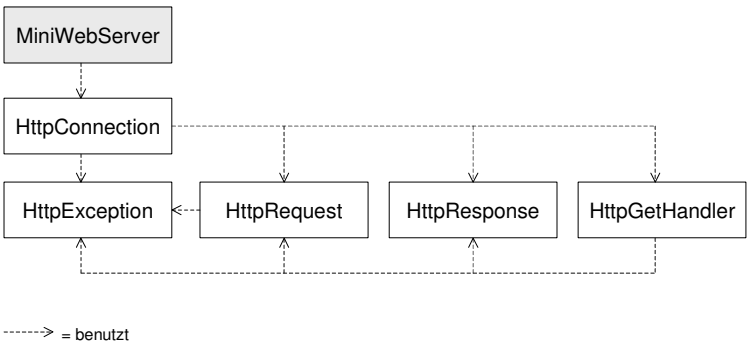


***Bild 5.6**: Die verwendeten Klassen*

Übersicht über die Funktionalität der Klassen:

`MiniWebServer`	Server starten Verbindungen akzeptieren Server herunterfahren
`HttpConnection`	Anfrage bearbeiten Ausnahmen behandeln
`HttpException`	Ausnahme, über die der Webbrowser informiert wird
`HttpRequest`	Anfrage einlesen und analysieren
`HttpResponse`	Kopfzeile und Antwortparameter zusammenstellen und senden
`HttpGetHandler`	angeforderte Ressource lokalisieren und senden

MiniWebServer

```
package webserver;

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

public class MiniWebServer {
  private int port;
  private String dir;
  private ServerSocket server;

  public MiniWebServer(int port, String dir) {
    this.port = port;
    this.dir = dir;
  }

  public void startServer() {
    System.setSecurityManager(new SecurityManager());

    // ShutdownHook registrieren
    Runtime.getRuntime().addShutdownHook(new Thread() {
      public void run() {
        terminate();
      }
    });

    try {
      FileInputStream in = new FileInputStream("alias.properties");
      Properties alias = new Properties();
      alias.load(in);
      in.close();

      server = new ServerSocket(port);

      InetAddress addr = InetAddress.getLocalHost();
      System.out.println("MiniWebServer auf " +
        addr.getHostName() + "/" + addr.getHostAddress() +
        ":" + port + " gestartet ...");

      while (true) {
        Socket client = server.accept();
        new HttpConnection(client, dir, alias).start();
      }
    }
    catch (SocketException e) { }
    catch (IOException e) {
      System.err.println(new Date() + " MiniWebServer: " + e);
    }
  }

  // ServerSocket schließen
  public void terminate() {
    try {
      if (server != null)
        server.close();
    }
```

```
    catch (IOException e) { }
    System.out.println(new Date() + ": MiniWebServer gestoppt");
  }

  public static void main(String[] args) {
    if (args.length != 2) {
      System.err.println("java MiniWebServer <port> <dir>");
      System.exit(1);
    }

    int port = Integer.parseInt(args[0]);
    String dir = args[1];
    new MiniWebServer(port, dir).startServer();
  }
}
```

Die Methode `startServer` registriert einen *Shutdownhook*, der den Server-Socket mit Hilfe der Methode `terminate` bei der Terminierung des Programms (z.B. durch Abbruch mit `Strg+C`) schließt. Hiermit wird der Einsatz eines Shutdownhooks demonstriert, er ist für die Lauffähigkeit des Programms in diesem Fall jedoch nicht erforderlich.

Wird der Server-Socket geschlossen, während noch die `ServerSocket`-Methode `accept` auf einen Verbindungswunsch wartet, so wird eine `SocketException` ausgelöst.

HttpConnection

```
package webserver;

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

public class HttpConnection extends Thread {
  private Socket client;
  private String dir;
  private Properties alias;
  private BufferedReader in;
  private OutputStream out;
  private HttpRequest request;
  private HttpResponse response;

  public HttpConnection(Socket client, String dir, Properties alias) {
    this.client = client;
    this.dir = dir;
    this.alias = alias;
  }

  public void run() {
    try {
      in = new BufferedReader(new InputStreamReader(
        client.getInputStream(), "ISO-8859-1"));
      out = client.getOutputStream();

      request = new HttpRequest(in);
```

```
      response = new HttpResponse(out);

      request.getRequest();

      String method = request.getMethod();
      String url = request.getURL();

      String clientAddr = client.getInetAddress().getHostAddress();
      int clientPort = client.getPort();
      System.out.println(new Date() + ": " + clientAddr + ":" +
        clientPort + " " + method + " " + url);

      // printHeaders();

      if (!method.equals("GET"))
        throw new HttpException(
          "Nur die GET-Methode ist zulässig", 501);

      // GET-Request verarbeiten
      new HttpGetHandler(dir, alias, request, response).process();
    }
    catch (HttpException e) {
      try {
        handleException(e);
      }
      catch (IOException ex) { }
    }
    catch (IOException e) {
      System.err.println(new Date() + " HttpConnection: " + e);
    }
    finally {
      try {
        if (client != null)
          client.close();
      }
      catch (IOException e) { }
    }
  }

  private void printHeaders() {
    Hashtable<String, String> headers = request.getHeaders();
    Enumeration<String> e = headers.keys();
    while (e.hasMoreElements()) {
      String key = e.nextElement();
      System.out.println("\t" + key + ": " + headers.get(key));
    }
  }

  private void handleException(HttpException e) throws IOException {
    int status = e.getStatus();
    response.printStatus(status);
    response.addHeader("Content-Type", "text/html");
    response.printHeaders();

    String statusMsg = status  + " " + response.getStatusText(status);
    StringBuilder msg = new StringBuilder();
    msg.append("<html><head><title>" + statusMsg + "</title></head>");
```

```
    msg.append("<body><h1>" + statusMsg + "</h1>" + e.getMessage()
      + "</body></html>");

    out.write(msg.toString().getBytes("ISO-8859-1"));
    out.flush();
  }
}
```

Ein `HttpRequest`- und ein `HttpResponse`-Objekt werden mit Hilfe des Socket-Eingabe- bzw. -Ausgabestroms erzeugt.

Da die Daten der HTTP-Anfrage gemäß "ISO Latin Alphabet No. 1" codiert sind, wird für den `InputStreamReader` der Zeichensatz `ISO-8859-1` zugrunde gelegt.

Die HTTP-Anfrage wird eingelesen. Die HTTP-Methode und der URL der Ressource werden ermittelt und protokolliert. Andere Methoden als `GET` führen zur Auslösung einer `HttpException`. Mit Hilfe eines `HttpGetHandler`-Objekts wird die Anfrage bearbeitet.

Evtl. Ausnahmen vom Typ `HttpException` werden von der Methode `handleException` behandelt. Zeichenketten werden für die Ausgabe in Bytes gemäß `ISO-8859-1` gewandelt.

HttpException

```
package webserver;

public class HttpException extends Exception {
  private int status;

  public HttpException(String msg, int status) {
    super(msg);
    this.status = status;
  }

  public int getStatus() {
    return status;
  }
}
```

HttpRequest

```
package webserver;

import java.io.*;
import java.util.*;

public class HttpRequest {
  private BufferedReader in;
  private String method = "";
  private String url = "";
  private String path = "";
  private String query = "";
  private Hashtable<String, String> headers;
```

```
public HttpRequest(BufferedReader in) {
  this.in = in;
  headers = new Hashtable<String, String>();
}

public String getMethod() {
  return method;
}

public String getURL() {
  return url;
}

public String getPath() {
  return path;
}

public String getQuery() {
  return query;
}

public Hashtable<String, String> getHeaders() {
  return headers;
}

public String getHeader(String key) {
  return headers.get(key);
}

public void getRequest() throws HttpException {
  try {
    // Erste Zeile (Request-Line) analysieren
    String line = in.readLine();
    if (line == null)
      return;

    StringTokenizer st = new StringTokenizer(line);
    method = st.nextToken();
    url = st.nextToken();

    // Path und Query String speichern
    path = url;
    int idx = url.indexOf('?');
    if (idx >= 0) {
      path = url.substring(0, idx);
      if (idx + 1 < url.length()) {
        query = url.substring(idx + 1);
      }
    }

    // Header analysieren und speichern
    while ((line = in.readLine()) != null && line.length() > 0) {
      int i = line.indexOf(':');
      if (i > 0) {
        String key = line.substring(0, i);
        String value = line.substring(i + 2);
```

```
          if (key != null && value != null) {
            headers.put(key.toLowerCase(), value);
          }
        }
      }
    }
    catch (IOException e) {
      throw new HttpException("Fehler beim Lesen des Request", 500);
    }
    catch (Exception e) {
      throw new HttpException("Fehler beim Parsen des Request", 400);
    }
  }
}
```

Die Methode `getRequest` liest die Zeilen der HTTP-Anfrage bis zur Leerzeile, ermittelt die HTTP-Methode, den URL, den Ressourcenpfad und evtl. den Query String. Anfrageparametername und -wert werden in einer *Hashtable* gespeichert. Diese verschiedenen Bestandteile können über `get`-Methoden abgefragt werden.

HttpResponse

```
package webserver;

import java.io.*;
import java.util.*;

public class HttpResponse {
  private Hashtable<String, String> headers;
  private OutputStream out;

  public HttpResponse(OutputStream out) {
    this.out = out;
    headers = new Hashtable<String, String>();
  }

  public OutputStream getOutputStream() {
    return out;
  }

  public void addHeader(String key, String value) {
    if (key != null && value != null) {
      headers.put(key, value);
    }
  }

  public String getStatusText(int status) {
    String txt;
    switch (status) {
      case 200:   txt = "OK";
            break;
      case 400:   txt = "Bad Request";
            break;
      case 404:   txt = "Not Found";
            break;
```

```
      case 500:   txt = "Internal Server Error";
            break;
      case 501:   txt = "Not Implemented";
            break;
      default:  txt = "";
    }
    return txt;
  }

  public void printStatus(int status) throws IOException {
    String line = "HTTP/1.0 " + status + " " + getStatusText(status)
      + "\r\n";
    out.write(line.getBytes("ISO-8859-1"));
    out.flush();
  }

  public void printHeaders() throws IOException {
    Enumeration<String> e = headers.keys();
    while (e.hasMoreElements()) {
      String key = e.nextElement();
      String line = key + ": " + headers.get(key) + "\r\n";
      out.write(line.getBytes("ISO-8859-1"));
    }

    String line = "\r\n";
    out.write(line.getBytes("ISO-8859-1"));
    out.flush();
  }
}
```

Antwortparameter werden in einer *Hashtable* verwaltet und können mit `addHeader` aufgenommen werden. `printStatus` schreibt die Kopfzeile und `printHeaders` die Antwortparameter in den Ausgabestrom.

HttpGetHandler

```
package webserver;

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

public class HttpGetHandler {
  private String dir;
  private Properties alias;
  private HttpRequest request;
  private HttpResponse response;

  public HttpGetHandler(String dir, Properties alias,
    HttpRequest request, HttpResponse response) {

    this.dir = dir;
    this.alias = alias;
    this.request = request;
    this.response = response;
  }
```

```
public void process() throws IOException, HttpException {
  String url = request.getURL();
  String path = url;

  int idx = url.indexOf('?');
  if (idx >= 0)
    throw new HttpException(
       "Query String wird nicht unterstützt", 501);

  if (path.endsWith("/"))
    path = path + "index.html";

  String filename = getFilename(path);
  File file = new File(filename);

  FileInputStream fis = null;
  try {
    fis = new FileInputStream(file);

    String type = URLConnection.getFileNameMap().getContentTypeFor(
       filename);
    if (type == null) {
      type = "application/octet-stream";
    }
    response.addHeader("Content-Type", type);
    response.addHeader("Content-Length",
       String.valueOf(file.length()));
    response.printStatus(200);
    response.printHeaders();

    OutputStream out = response.getOutputStream();
    byte data[] = new byte[1024];
    int cnt;
    while ((cnt = fis.read(data)) != -1) {
      out.write(data, 0, cnt);
    }
    out.flush();
  }
  catch (SecurityException e) {
    throw new HttpException(e.getMessage(), 400);
  }
  catch (FileNotFoundException e) {
    throw new HttpException(e.getMessage(), 404);
  }
  finally {
    try {
      if (fis != null)
        fis.close();
    }
    catch (IOException e) { }
  }
}

private String getFilename(String path) {
  int idx = path.indexOf("/", 1);

  String filename;
```

```
    if (idx > 0) {
      String key = path.substring(1, idx);
      String value = alias.getProperty(key);
      if (value == null)
        filename = dir + path;
      else
        filename = value + path.substring(idx);
    }
    else
      filename = dir + path;

    return filename;
  }
}
```

`process` führt die Aktion `GET` aus. Query Strings werden nicht unterstützt. Endet der Name der Ressource mit einem Schrägstrich `/`, so wird standardmäßig nach der HTML-Datei `index.html` in dem so bezeichneten Verzeichnis gesucht. Die einzigen Antwortparameter sind `Content-Type` und `Content-Length`.

Die Methode `getFilename` ermittelt den Dateinamen der angeforderten Ressource. Hierbei werden auch evtl. angegebene virtuelle Verzeichnisse (wie eingangs beschrieben) berücksichtigt.

Der MIME-Typ der Ressource wird mit der Methode

```
URLConnection.getFileNameMap().getContentTypeFor(...)
```

ermittelt.

Die abstrakte Klasse `java.net.URLConnection` ist die Superklasse derjenigen Klassen, die eine Netzverbindung zu einer durch den URL adressierten Ressource repräsentieren.

`static FileNameMap` **`getFileNameMap`**`()`
lädt die Tabelle der MIME-Typen aus einer Datei (*lib\content-types.properties* in dem durch die System Property `"java.home"` bezeichneten Verzeichnis).

Das Interface `java.net.FileNameMap` enthält die Methode

```
String getContentTypeFor(String fileName).
```

Sie liefert den MIME-Typ zu einem Dateinamen.

5.5 Webseiten dynamisch erzeugen

Programm 5.5

Wir erweitern nun den Mini-Webserver aus Kapitel 5.4 um zusätzliche Funktionen.

So können dann z.B. Formulare ausgewertet, Daten in einer Datenbank abgespeichert oder abgefragt werden. Zu diesem Zweck unterstützt der neue Server auch die `POST`-Methode.

Die neue Anwendungsfunktionalität zur dynamischen Erzeugung von HTML-Seiten wird in eigenen Klassen (im Weiteren *Anwen-*

dungsmodule genannt) – unabhängig vom Webserver – realisiert. Diese Klassen werden nach Bedarf zur Laufzeit des Webservers dynamisch hinzu geladen.

Die Klassen `XMiniWebServer` und `MiniWebServer` (aus Kapitel 5.4) sind mit Ausnahme des Namens und der `package`-Klausel identisch.

XMiniWebServer

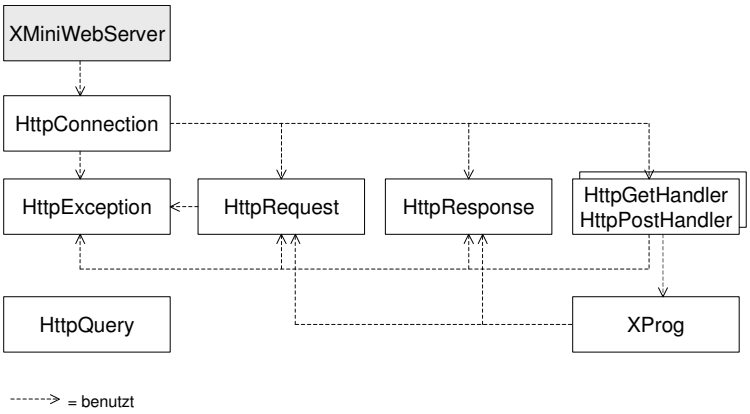


***Bild 5.7**: Die Klassen des erweiterten Webservers*

Wie `HttpConnection` aus Kapitel 5.4, aber mit Unterstützung der `POST`-Methode:

HttpConnection

```
// Nur GET und POST sind zulässig
if (method.equals("GET"))
  (new HttpGetHandler(dir, alias, request, response)).process();
else if (method.equals("POST"))
  (new HttpPostHandler(request, response)).process();
else
  throw new HttpException("Nur GET und POST sind zulässig", 501);
```

`HttpException` entspricht der gleichnamigen Klasse aus Kapitel 5.4.

HttpException

Hier sind zwei neue Methoden vorhanden:

HttpRequest

```
public int getContentLength() {
  String value = getHeader("content-length");
  if (value != null)
    return Integer.parseInt(value);
  else
    return -1;
}

public BufferedReader getBufferedReader() {
  return in;
}
```

HttpResponse

`HttpResponse` entspricht der gleichnamigen Klasse aus Kapitel 5.4.

HttpGetHandler

Gegenüber der Version aus Kapitel 5.4 enthält diese Klasse eine geänderte Methode `process` und die neue Methode `execute`. `process` behandelt Ressourcennamen der Form `/prog/Xxx` auf besondere Weise. `Xxx` wird als Anwendungsmodul im oberen Sinne betrachtet. Der vollständige Klassenname ist dann `prog.Xxx`.

```
public void process() throws IOException, HttpException {
  String path = request.getPath();
  String query = request.getQuery();

  // Ein Pfad der Form "/prog/Xxx" führt zum Laden der Klasse prog.Xxx

  int idx = path.lastIndexOf('/');
  String s = path.substring(0, idx);

  if (s.equals("/prog")) {
    String classname = path.substring(1).replace('/', '.');
    execute(classname);
    return;
  }

  if (query.length() > 0)
    throw new HttpException("Programmpfad unbekannt", 404);

  if (path.endsWith("/"))
    path = path + "index.html";

  String filename = getFilename(path);
  File file = new File(filename);

  FileInputStream fis = null;
  try {
    fis = new FileInputStream(file);

    String type = URLConnection.getFileNameMap().getContentTypeFor(
      filename);
    if (type == null) {
      type = "application/octet-stream";
    }
    response.addHeader("Content-Type", type);
    response.addHeader("Content-Length",
    String.valueOf(file.length()));
    response.printStatus(200);
    response.printHeaders();

    OutputStream out = response.getOutputStream();
    byte data[] = new byte[1024];
    int cnt;
    while ((cnt = fis.read(data)) != -1) {
      out.write(data, 0, cnt);
    }
```

```
    out.flush();
  }
  catch (SecurityException e) {
    throw new HttpException(e.getMessage(), 400);
  }
  catch (FileNotFoundException e) {
    throw new HttpException(e.getMessage(), 404);
  }
  finally {
    try {
      if (fis != null)
        fis.close();
    }
    catch (IOException e) { }
  }
}

private void execute(String classname) throws IOException,
  HttpException {
  try {
    // Klasse laden
    Class c = Class.forName(classname);

    // Methode "getInstance" bereitstellen
    Class[] types = {};
    Method m = c.getMethod("getInstance", types);

    // getInstance aufrufen
    Object[] args = {};
    XProg p = (XProg) m.invoke(null, args);

    p.doGet(request, response);
  }
  catch (ClassNotFoundException e) {
    throw new HttpException(e.getMessage(), 404);
  }
  catch (Exception e) {
    throw new HttpException(e.getMessage(), 500);
  }
}
```

Die Klasse `prog.Xxx` wird geladen. Anwendungsmodule sind Subklassen der abstrakten Klasse `XProg` und enthalten die statische Methode `getInstance` (ohne Parameter) sowie die Methoden `doGet` und `doPost`.

Die `Class`-Methode

```
Method getMethod(String name, Class... parameterTypes)
    throws NoSuchMethodException
```

liefert ein `Method`-Objekt. `name` ist der Name der gewünschten `public`-Methode, `parameterTypes` sind die Parametertypen dieser Methode, durch `Class`-Objekte repräsentiert.

Die Klasse `java.lang.reflect.Method` repräsentiert eine Methode.

Die `Method`-Methode

```
Object invoke(Object obj, Object... args)
    throws IllegalAccessException,
    java.lang.reflect.InvocationTargetException
```

ruft die durch dieses `Method`-Objekt repräsentierte Methode auf. `obj` ist das Objekt, für das die Methode ausgeführt werden soll, `args` sind die Argumente für den Methodenaufruf. Bei einfachen Datentypen werden hier die entsprechenden Hüllobjekte eingesetzt. Ist die Methode statisch, so wird `obj` ignoriert und kann `null` sein.

In unserem Fall wird die Klassenmethode `getInstance` des geladenen Anwendungsmoduls aufgerufen. Sie liefert eine Instanz des Anwendungsmoduls vom Typ `XProg`. Hierfür wird nun `doGet` aufgerufen.

Dieser *indirekte Mechanismus* zum Laden von Klassen und Aufruf von Methoden ermöglicht es, dass die Klasse zum Zeitpunkt der Compilierung nicht bekannt sein muss.

HttpPostHandler

```
package xwebserver;

import java.io.*;
import java.lang.reflect.*;

public class HttpPostHandler {
  private HttpRequest request;
  private HttpResponse response;

  public HttpPostHandler(HttpRequest request, HttpResponse response) {
    this.request = request;
    this.response = response;
  }

  public void process() throws IOException, HttpException {
    String path = request.getPath();

    // Ein Pfad der Form "/prog/Xxx" führt zum Laden der Klasse
    // prog.Xxx

    int idx = path.lastIndexOf('/');
    String s = path.substring(0, idx);

    if (s.equals("/prog")) {
      String classname = path.substring(1).replace('/', '.');
      execute(classname);
      return;
    }
    else
      throw new HttpException("Programmpfad unbekannt", 404);
  }
```

```
  private void execute(String classname) throws IOException,
    HttpException {

    try {
      // Klasse laden
      Class c = Class.forName(classname);

      // Methode "getInstance" bereitstellen
      Class[] types = {};
      Method m = c.getMethod("getInstance", types);

      // getInstance aufrufen
      Object[] args = {};
      XProg p = (XProg) m.invoke(null, args);

      p.doPost(request, response);
    }
    catch (ClassNotFoundException e) {
      throw new HttpException(e.getMessage(), 404);
    }
    catch (Exception e) {
      throw new HttpException(e.getMessage(), 500);
    }
  }
}
```

Anwendungsmodule müssen von der abstrakten Klasse `XProg` abgeleitet sein. `XProg` implementiert so genannte Default-Methoden, die ausgeführt werden, falls das Anwendungsmodul die entsprechende Methode nicht überschreibt.

XProg

```
package xwebserver;

import java.io.*;

public abstract class XProg {
  public void doGet(HttpRequest request, HttpResponse response)
    throws IOException {

    response.addHeader("Content-Type", "text/html");
    response.printStatus(200);
    response.printHeaders();

    int status = 501;
    String statusMsg = status  + " " + response.getStatusText(status);
    StringBuilder msg = new StringBuilder();
    msg.append("<html><head><title>" + statusMsg + "</title></head>");
    msg.append("<body><h1>GET nicht unterstützt</h1></body></html>");

    OutputStream out = response.getOutputStream();
    out.write(msg.toString().getBytes("ISO-8859-1"));
    out.flush();
  }
```

```
  public void doPost(HttpRequest request, HttpResponse response)
    throws IOException {

    response.addHeader("Content-Type", "text/html");
    response.printStatus(200);
    response.printHeaders();

    int status = 501;
    String statusMsg = status  + " " + response.getStatusText(status);
    StringBuilder msg = new StringBuilder();
    msg.append("<html><head><title>" + statusMsg + "</title></head>");
    msg.append("<body><h1>POST nicht unterstützt</h1></body></html>");

    OutputStream out = response.getOutputStream();
    out.write(msg.toString().getBytes("ISO-8859-1"));
    out.flush();
  }
}
```

HttpQuery

Die Klasse `HttpQuery` enthält Methoden, die Parameterwerte eines *Query Strings* in decodierter Form bereitstellen. Zu beachten ist, dass ein Parametername in einem Query String auch mehrfach auftreten kann. Parametername und -wert werden in einer *Hashtable* gespeichert. Der Parametername stellt den Schlüssel dar, die evtl. mehrfach auftretenden Parameterwerte zum gleichen Namen werden in einem Vektor zusammengefasst, der dann als Wert zum Schlüssel in die *Hashtable* eingetragen wird.

```
package xwebserver;

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

public class HttpQuery {
  private Hashtable<String, Vector<String>> h;

  public HttpQuery(String query) {
    h = new Hashtable<String, Vector<String>>();
    parseQueryString(query);
  }

  public String getParameter(String name) {
    Vector<String> v = h.get(name);
    if (v == null)
      return null;
    else
      return v.get(0);
  }

  public String[] getParameterValues(String name) {
    Vector<String> v = h.get(name);
    if (v == null)
      return null;
```

```
    else {
      String[] result = new String[v.size()];
      v.copyInto(result);
      return result;
    }
  }

  /*  Allgemeine Form eines Query Strings:
      name1=value1&name2=value2...
      Ein Feldname kann auch mehrfach auftreten. */

  private void parseQueryString(String query) {
    StringTokenizer params = new StringTokenizer(query, "&");
    StringTokenizer param;
    String name, value;

    while (params.hasMoreTokens()) {
      param = new StringTokenizer(params.nextToken(), "=");
      name = param.nextToken();
      if (param.hasMoreTokens())
        value = param.nextToken();
      else
        value = "";

      if (name != null) {
        String dvalue = "";
        try {
          dvalue = URLDecoder.decode(value, "ISO-8859-1");
        }
        catch (UnsupportedEncodingException e) { }

        // Prüfen, ob der Name bereits in der Hashtable vorkommt.
        // Die Werte werden jeweils in einem Vektor gespeichert.

        Vector<String> v = h.get(name);
        if (v == null) {
          v = new Vector<String>();
          v.add(dvalue);
          h.put(name, v);
        }
        else {
          v.add(dvalue);
        }
      }
    }
  }
}
```

Das *Anwendungsmodul* `Test` soll die neuen Funktionen demonstrieren. `Test` wertet ein Formular aus.

post.html

```
<html>
<head><title>Test (POST)</title></head>
<body>
  <form action="/prog/Test" method="POST">
    Name: <input type="text" size="60" name="name"/><p/>
    <input type="checkbox" name="sprache" value="Java"/>Java<br/>
    <input type="checkbox" name="sprache" value="C++"/>C++<br/>
    <input type="checkbox" name="sprache" value="C#"/>C#<p/>
    <input type="submit" value="Senden"/>
    <input type="reset" value="Löschen"/>
  </form>
</body>
</html>
```

Test

```
package prog;

import java.io.*;
import java.util.*;
import xwebserver.*;

public class Test extends XProg {
  private static final XProg INSTANCE = new Test();

  private Test() { }

  public static XProg getInstance() {
    return INSTANCE;
  }

  public void doGet(HttpRequest request, HttpResponse response)
    throws IOException {

    String q = request.getQuery();
    process(request, response, q);
  }

  public void doPost(HttpRequest request, HttpResponse response)
    throws IOException {

    BufferedReader in = request.getBufferedReader();
    int length = request.getContentLength();

    int c;
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < length; i++) {
      c = in.read();
      if (c == -1)
        break;
      sb.append((char) c);
    }

    String q = sb.toString();
    process(request, response, q);
  }
```

```
  private void process(HttpRequest request, HttpResponse response,
    String q) throws IOException {

    HttpQuery query = new HttpQuery(q);
    String name = query.getParameter("name");
    String[] sprachen = query.getParameterValues("sprache");

    response.addHeader("Content-Type", "text/html");
    response.printStatus(200);
    response.printHeaders();

    StringBuilder buffer = new StringBuilder();
    buffer.append("<html><head><title>Test</title></head>");
    buffer.append("<body><h1>Request Header</h1>");

    Hashtable<String, String> headers = request.getHeaders();
    Enumeration<String> e = headers.keys();
    while (e.hasMoreElements()) {
      String key = e.nextElement();
      buffer.append(key + ": " + headers.get(key) + "<br/>");
    }

    buffer.append("<h1>Formularparameter</h1>");
    buffer.append("Name: " + name + "<p/>");
    buffer.append("Programmiersprachen:<br/>");
    if (sprachen != null) {
      for (int i = 0; i < sprachen.length; i++)
        buffer.append(sprachen[i] + "<br/>");
    }
    buffer.append("</body></html>");

    OutputStream out = response.getOutputStream();
    out.write(buffer.toString().getBytes("ISO-8859-1"));
    out.flush();
  }
}
```

Die Klasse `Test` ist ein so genannter *Singleton*, d.h. sie wird genau einmal instanziiert. Bei jedem Aufruf wird die vorhandene Instanz zurückgeliefert, so dass also zur Laufzeit des Servers höchstens ein Objekt dieser Klasse vorhanden ist.

***Bild 5.8**: Formular post.html*

Bild 5.9: Ergebnis

Die Anwendung der `GET`-Methode (im Formular ist `POST` durch `GET` zu ersetzen) führt zu einem entsprechenden Ergebnis.

5.6 Aufgaben

1. Entwickeln Sie ein Programm, mit dem Dateien vom Webserver mittels HTTP herunter geladen werden können. Nutzen Sie hierzu die Socket-Programmierung mit TCP (siehe Kapitel 4).

2. Realisieren Sie eine Variante zum Downloadprogramm in Aufgabe 1, indem Sie die folgenden Methoden der Klassen `URL` und `URLConnection` benutzen.

   Die Klasse `java.net.URL` repräsentiert einen *Uniform Resource Locator*.

   `URL``(String spec) throws MalformedURLException`
   erzeugt ein `URL`-Objekt aus der Zeichenkette `spec`. Die Klasse `java.net.MalformedURLException` ist von `java.io.IOException` abgeleitet.

   Die Klasse `URL`-Methode
   `URLConnection` **`openConnection`**`() throws IOException`
   liefert ein `java.net.URLConnection`-Objekt, das eine Verbindung zu einer durch den URL adressierten Ressource repräsentiert.

`URLConnection`-Methoden:

`void` **`connect`**`() throws IOException`
stellt einer Verbindung zu der durch den URL adressierten Ressource her.

`int` **`getContentLength`**`()`
liefert den Wert des Antwortparameters `Content-Length`.

`InputStream` **`getInputStream`**`() throws IOException`
liefert einen Eingabestrom zum Lesen über diese Verbindung.

3. Entwickeln Sie für den erweiterten Webserver `XMiniWebServer` aus Kapitel 5.5 ein *Anwendungsmodul* `Buecherliste`, das zu einem vom Benutzer in einem Formular eingegebenen Wort Angaben zu denjenigen Büchern aus einer Bücher-Datenbank anzeigt, deren Titel dieses Wort enthalten.

   Hierzu ist die Datei *java.policy* anzupassen. Das Programm muss eine Socket-Verbindung zum Port des Datenbankservers aufnehmen können (bei MySQL Portnummer 3306) und Leserecht für den JDBC-Treiber haben, z. B.

   ```
   permission java.io.FilePermission " C:\\Programme\\java\\-",
     "read";
   ```

   (falls sich der JDBC-Treiber im Verzeichnis <Java-Inst>\lib\ext befindet).

4. Entwickeln Sie eine Variante des Webservers aus Kapitel 5.4, die einen Thread-Pool nutzt (siehe Kapitel 4.8).

   In der Datei *java.policy* muss die folgende Regel zusätzlich eingefügt werden:

   ```
   permission java.lang.RuntimePermission "modifyThread";
   ```

6 XML Remote Procedure Calls (XML-RPC)

Eine bewährte und sehr verbreitete Technik zur Entwicklung von Client-Server-Anwendungen ist der so genannte *Remote Procedure Call* (RPC). Der Client ruft mit einem lokalen Prozeduraufruf einen Dienst auf, der in der Regel auf einem anderen Rechner angeboten wird. Implementierungen des RPC-Mechanismus unterscheiden sich im Allgemeinen für verschiedene Programmiersprachen.

In diesem Kapitel wird am Beispiel von *XML-RPC* gezeigt, wie das Protokoll HTTP und XML als Kommunikationsformat für den entfernten Prozeduraufruf eingesetzt werden können. Vom Konzept her ist diese Technik programmiersprachenunabhängig. Wir werden uns hauptsächlich mit der Java-Implementierung beschäftigen. Die Aufgabe 5 dieses Kapitels nutzt eine PHP-Implementierung auf der Client-Seite.

6.1 Grundkonzept und ein erstes Beispiel

XML-RPC ist ein einfaches Protokoll für den entfernten Prozeduraufruf (RPC). XML-RPC verwendet XML als Datenaustauschformat. *Anfrage* (Methodenname, Parameter) und *Antwort* (Rückgabewert des Methodenaufrufs) werden als XML-Dokumente mit HTTP übermittelt.

Bild 6.1 zeigt den Nutzdatenteil der HTTP-Anfrage (HTTP-Methode ist `POST`) und den Nutzdatenteil der HTTP-Antwort beim Aufruf der entfernten Methode `getEcho`.

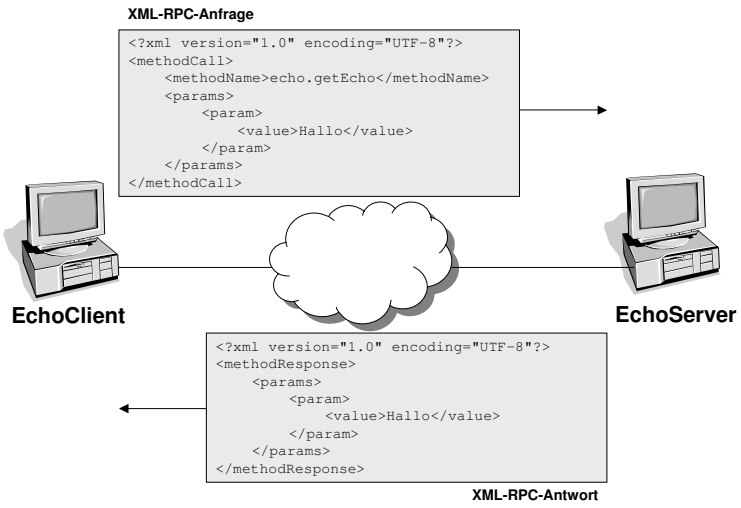


***Bild 6.1**: XML-RPC-Anfrage und Antwort*

Die Nutzdatenteile werden auch als *XML-RPC-Anfrage* bzw. *-Antwort* bezeichnet.

Eine XML-RPC-Anfrage kann *mehrere* Parameterwerte enthalten (gemäß der Signatur der entfernten Methode). Eine XML-RPC-Antwort enthält jedoch *genau einen* Wert.

Struktur der XML-RPC-Anfrage

```
<methodCall>
  <methodName>Methodenname</methodName>
  <params>
    <param>
      <value>Parameterwert</value>
    </param>
    ...
  </params>
</methodCall>
```

Struktur der XML-RPC-Antwort

```
<methodResponse>
  <params>
    <param>
      <value>Rückgabewert</value>
    </param>
  </params>
</methodResponse>
```

Datentypen

Parameter- und Rückgabewerte haben jeweils einen bestimmten Datentyp. XML-RPC unterstützt insgesamt acht Datentypen: *einfache Datentypen* (z.B. Zeichenketten, Zahlen) und *zusammengesetzte Datentypen* (z.B. Arrays).

Die Datentypen werden im nächsten Abschnitt einzeln anhand von Programmbeispielen vorgestellt.

Spezifikation

Die *Spezifikation von XML-RPC* wurde 1999 von Dave Winer veröffentlicht. Sie umfasst nur wenige Seiten und kann unter der Adresse

```
http://www.xmlrpc.com
```

nachgeschlagen werden.

Implementierungen von XML-RPC existieren für verschiedene Programmier- und Skriptsprachen (z.B. Java, Perl, PHP, Python). Somit kann beispielsweise ein PHP-Client mit einem Java-Server kommunizieren (siehe Aufgabe 5).

Apache XML-RPC

Wir nutzen im Folgenden die Java-Implementierung *Apache XML-RPC* der Apache Software Foundation in der Version 3.0. Die erforderlichen Klassenbibliotheken (JAR-Dateien) können von der Website

`http://ws.apache.org/xmlrpc/`

heruntergeladen werden.

Die JAR-Dateien können in ein Verzeichnis Ihrer Wahl kopiert werden (siehe auch die Programmsammlung zu diesem Buch).

Die Version 3.0 von Apache XML-RPC erfüllt die offizielle XML-RPC-Spezifikation, bietet aber auch Erweiterungen (Unterstützung zusätzlicher Datentypen, Performancesteigerung durch Nutzung des so genannten Streaming-Verfahrens, Datenkompression), die in einer reinen Java-Umgebung sinnvoll eingesetzt werden können. Wir beschränken uns hier hauptsächlich auf die Vorstellung der zur XML-RPC-Spezifikation kompatiblen Aspekte. Lediglich Programm 6.10 geht in einem Aspekt auf die Erweiterungen ein.

SOAP

Eine Weiterentwicklung von XML-RPC ist *SOAP*, das vom World Wide Web Consortium (W3C) definiert wird und neben anderen Standardtechnologien zur Entwicklung von so genannten *Web Services* genutzt wird. SOAP war ehemals Abkürzung für "Simple Object Access Protocol" und ist jetzt ein eigenständiger Name.

Im Vergleich zu SOAP hat XML-RPC einen geringeren Leistungsumfang, ist aber auch wesentlich einfacher in der Handhabung. Für viele Anwendungen erweist sich XML-RPC als völlig ausreichend und kann auch für die Implementierung von Web Services eingesetzt werden.

WebServer und XmlRpcServer

Zur Implementierung eines *XML-RPC-Servers* stellt Apache XML-RPC die Klassen

`org.apache.xmlrpc.webserver.WebServer` und
`org.apache.xmlrpc.server.XmlRpcServer`

zur Verfügung.

`WebServer` implementiert einen speziell für die Behandlung von XML-RPC-Anfragen geeigneten HTTP-Server, der in eigene Applikationen eingebettet werden kann. Dieser ist zu Testzwecken sehr gut geeignet. Für höhere Ansprüche in Bezug auf Performance und Stabilität sind ausgereifte Servlet-Container wie beispielsweise Apache Tomcat besser geeignet.

Die Klasse `XmlRpcServer` verarbeitet die XML-RPC-Anfragen.

`WebServer``(int port)`
erzeugt einen Server mit der Portnummer `port`.

`WebServer`-Methoden:

`void` **`start`**`() throws java.io.IOException`
startet den Server.

`void` **`shutdown`**`()`
stoppt den Server.

Die Methode **`getXmlRpcServer`**`()` liefert ein `XmlRpcServer`-Objekt.

Handler

Ein so genannter *Handler* ist eine entfernt aufrufbare Methode. Folgende Bedingungen müssen erfüllet sein:

- Die Methode muss `public` sein, nicht `static` sein und darf nicht den Rückgabetyp `void` haben.
- In der Klasse, die den Handler implementiert, muss der parameterlose Standardkonstruktor implizit oder explizit vorhanden sein.

Die Klasse `org.apache.xmlrpc.server.PropertyHandlerMapping` kann mehrere Handler verwalten.

Ihre Methode

`void` **`addHandler`**`(String key, Class typ) throws XmlRpcException`

fügt die Handler-Klasse `typ` mit dem Namen `key` hinzu. Hierzu wird das `Class`-Objekt der Handler-Klasse angegeben: `Klassenname.class`.

Eine Ausnahme vom Typ `org.apache.xmlrpc.XmlRpcException` wird generell ausgelöst, wenn der Server einen Fehler meldet.

`void` **`removeHandler`**`(String key)`
entfernt alle Handler mit dem Namen `key`.

Mit dem Aufruf der `XmlRpcServer`-Methode `setHandlerMapping` werden die Handler für das `XmlRpcServer`-Objekt registriert:

`setHandlerMapping``(propertyHandlerMapping)`

Programm 6.1

Das folgende Beispiel demonstriert eine einfache XML-RPC-Anwendung.

Die Klasse `Echo` implementiert die Methoden `getEcho` und `getEchoWithDate`, die ein "Echo" ohne bzw. mit Serverdatum zurückgeben.

Echo

```
import java.util.*;
import java.text.*;

public class Echo {
  public String getEcho(String s) {
    return s;
  }

  public String getEchoWithDate(String s) {
    SimpleDateFormat f = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
    return "[" + f.format(new Date()) + "] " + s;
  }
}
```

Die Klasse `EchoServer` erzeugt eine Instanz der Klasse `WebServer`, registriert den Echo-Dienst mit dem Namen "echo" und startet dann den Server.

EchoServer

```
import org.apache.xmlrpc.server.*;
import org.apache.xmlrpc.webserver.*;

public class EchoServer {
  public static void main(String[] args) throws Exception {
    int port = Integer.parseInt(args[0]);

    PropertyHandlerMapping phm = new PropertyHandlerMapping();
    phm.addHandler("echo", Echo.class);

    WebServer webServer = new WebServer(port);
    XmlRpcServer server = webServer.getXmlRpcServer();
    server.setHandlerMapping(phm);
    webServer.start();
  }
}
```

XmlRpcClient

Zur Implementierung eines *XML-RPC-Client* stellt Apache XML-RPC die Klasse

`org.apache.xmlrpc.client.XmlRpcClient`

zur Verfügung.

Mit Hilfe der Klasse

`org.apache.xmlrpc.client.XmlRpcClientConfigImpl`

kann ein `XmlRpcClient`-Objekt konfiguriert werden.

Die `XmlRpcClientConfigImpl`-Methode

`void` **`setServerURL`**`(java.net.URL url)`

legt den URL des Servers fest. Im Beispiel: `http://localhost:50000`.

Mit dem Aufruf der `XmlRpcClient`-Methode `setConfig` wird die Konfiguration `config` für den Client gesetzt: `setConfig(config)`.

execute

`Object` **`execute`**`(String method, Object[] params) throws XmlRpcException` erzeugt eine XML-RPC-Anfrage und sendet sie mittels HTTP zum Server. Die zurückgeschickte XML-RPC-Antwort wird geparst und als Objekt vom Typ `Object` zurückgegeben.

Der String `method` hat den Aufbau:

```
Dienstname.Methodenname
```

`Dienstname` ist der Name, unter dem der Dienst auf der Serverseite registriert ist. `Methodenname` ist der Name der Methode, die der Dienst implementiert hat. Das Array `params` enthält die erforderlichen Parameter.

Wie die Parameter passend zur Signatur der Methode erzeugt werden müssen, zeigen die nächsten Programmbeispiele.

`EchoClient` ruft beide Methoden des Echo-Dienstes mit dem Argument "Hallo" auf.

EchoClient

```
import java.net.*;
import org.apache.xmlrpc.client.*;

public class EchoClient {
  public static void main(String args[]) throws Exception {
    URL url = new URL(args[0]);
    XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
    config.setServerURL(url);
    XmlRpcClient client = new XmlRpcClient();
    client.setConfig(config);

    Object[] params = {"Hallo"};
    String s = (String) client.execute("echo.getEcho", params);
    System.out.println(s);

    String t = (String) client.execute(
      "echo.getEchoWithDate", params);
    System.out.println(t);
  }
}
```

Ablauf beim entfernten Methodenaufruf

Die einzelnen Schritte beim entfernten Methodenaufruf sind:

1. Das Client-Programm erzeugt eine `XmlRpcClient`-Instanz, konfiguriert sie und ruft dann die Methode `execute` mit Angabe des Dienstnamens, des Methodennamens und der Parameter auf.
2. Diese Angaben werden in ein XML-Dokument verpackt und per `HTTP-POST` an den Server geschickt.

3. Der Server empfängt die HTTP-Anfrage und leitet die Verarbeitung des XML-Dokuments ein.
4. Das XML-Dokument wird geparst und anschließend die angegebene Methode des Dienstes aufgerufen.
5. Die Methode übergibt das Ergebnis an den XML-RPC-Verarbeitungsprozess, der dann das Ergebnis in ein XML-Dokument verpackt.
6. Der Server schickt das XML-Dokument als Antwort auf die HTTP-Anfrage zurück.
7. Der XML-RPC-Client parst das XML-Dokument, extrahiert den Rückgabewert und übergibt diesen als Objekt an das Client-Programm.

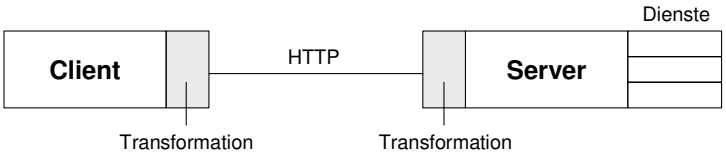


***Bild 6.2**: Kommunikation zwischen Client und Server*

Bevor das Werkzeug *Ant* mit *build.xml* aus der Programmsammlung genutzt werden kann, muss die Umgebungsvariable `XMLRPC_PATH` gesetzt werden. Hier müssen alle erforderlichen JAR-Dateien aufgeführt werden.

```
set XMLRPC_PATH=Verzeichnis/xxx.jar;...
```

1. Compilieren: `ant compile`
2. Server starten: `ant server`
3. Client starten: `ant client`

Ausgabe:

```
Hallo
[09.08.2006 14:15:33] Hallo
```

Verzichtet man auf die Nutzung des Werkzeugs *Ant*, so können obige drei Schritte wie folgt ausgeführt werden:

```
mkdir build

javac -sourcepath src -cp %XMLRPC_PATH% -d build src/*.java

start java -cp build;%XMLRPC_PATH% EchoServer 50000

java -cp build;%XMLRPC_PATH% EchoClient http://localhost:50000
```

6.2 XML-RPC-Datentypen

Zu den einfachen Datentypen gehören:

- Ganzzahl,
- Gleitkommazahl,
- Wahrheitswert,
- Zeichenkette,
- Datum/Zeit,
- Binärdaten.

Zusammengesetzte Datentypen sind:

- Array und
- Struktur.

Ein Array-Element kann einen Wert vom einfachen oder zusammengesetzten Datentyp enthalten.

Eine Struktur ist eine Folge von Elementen, die jeweils aus einem Namen und einem Wert bestehen. Der Name muss eine Zeichenkette sein, der Wert kann vom einfachen oder zusammengesetzten Datentyp sein.

Die folgende Tabelle gibt eine Übersicht.

Datentypen

XML-Tag-Name	Java-Typ für execute	Java-Typ für Handler
i4	java.lang.Integer	int
double	java.lang.Double	double
boolean	java.lang.Boolean	boolean
string	java.lang.String	java.lang.String
dateTime.iso8601	java.util.Date	java.util.Date
base64	byte[]	byte[]
array	java.lang.Object[]	java.lang.Object[]
struct	java.util.Map	java.util.Map

Die erste Tabellenspalte enthält die Namen der Tags für das vom XML-RPC-Protokoll verwendete XML-Dokument. Diese Tags markieren den Datenwert zum entsprechenden Datentyp. Die zweite Spalte enthält die Entsprechungen in Java für den Aufruf der `XmlRpcClient`-Methode `execute`. Die Datentypen der dritten Spalte werden als Parameter- bzw. Rückgabetypen der Handler benutzt. `void`-Methoden sind als Handler-Methoden nicht erlaubt. Der Wert `null` ist weder als Argumentwert noch als Rückgabewert erlaubt.

Mit Programm 6.2 kann die Handhabung der verschiedenen Datentypen getestet werden. ***Programm 6.2***

DatentypTestServer

```
import org.apache.xmlrpc.server.*;
import org.apache.xmlrpc.webserver.*;

public class DatentypTestServer {
  public static void main(String[] args) throws Exception {
    int port = Integer.parseInt(args[0]);

    PropertyHandlerMapping phm = new PropertyHandlerMapping();
    phm.addHandler("test", DatentypTest.class);

    WebServer webServer = new WebServer(port);
    XmlRpcServer server = webServer.getXmlRpcServer();
    server.setHandlerMapping(phm);
    webServer.start();
  }
}
```

Die Handler-Klasse `DatentypTest` enthält für jeden XML-RPC-Datentyp eine Testmethode. *DatentypTest*

```
import java.util.*;
import java.io.*;

public class DatentypTest {
  // Ganzzahl
  public int testInt(int x) {
    return x;
  }

  // Gleitkommazahl
  public double testDouble(double x) {
    return x;
  }

  // Wahrheitswert
  public boolean testBoolean(boolean x) {
    return x;
  }

  // Zeichenkette
  public String testString(String x) {
    return x;
  }

  // Datum/Zeit
  public Date testDateTime(Date x) {
    return x;
  }
```

```
  // Binärdaten
  public byte[] testBase64() throws IOException {
    InputStream in = new FileInputStream("java.gif");
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    int c;
    while ((c = in.read()) != -1) {
      out.write(c);
    }
    in.close();
    return out.toByteArray();
  }

  // Array
  public Object[] testArray(Object[] x) {
    return x;
  }

  // Struktur
  public Map testStruct(Map x) {
    return x;
  }
}
```

DatentypTestClient

Die Klasse `DatentypTestClient` enthält für jeden XML-RPC-Datentyp den Aufruf der zugehörigen Testmethode.

```
import java.util.*;
import java.io.*;
import java.net.*;
import org.apache.xmlrpc.client.*;

public class DatentypTestClient {
  public static void main(String args[]) throws Exception {
    URL url = new URL(args[0]);
    String typ = args[1];

    XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
    config.setServerURL(url);
    XmlRpcClient client = new XmlRpcClient();
    client.setConfig(config);

    // Ganzzahl
    if (typ.equals("int")) {
      System.out.println(typ);
      Object[] params = {1234};
      int r = (Integer) client.execute("test.testInt", params);
      System.out.println(r);
      System.out.println();
    }
    // Gleitkommazahl
    else if (typ.equals("double")) {
      System.out.println(typ);
      Object[] params = {123.456};
      double r = (Double) client.execute("test.testDouble", params);
```

```
    System.out.println(r);
    System.out.println();
  }
  // Wahrheitswert
  else if (typ.equals("boolean")) {
    System.out.println(typ);
    Object[] params = {true};
    boolean r = (Boolean) client.execute(
       "test.testBoolean", params);
    System.out.println(r);
    System.out.println();
  }
  // Zeichenkette
  else if (typ.equals("string")) {
    System.out.println(typ);
    Object[] params = {"<-- A & O -->"};
    String r = (String) client.execute("test.testString", params);
    System.out.println(r);
    System.out.println();
  }
  // Datum/Zeit
  else if (typ.equals("dateTime")) {
    System.out.println(typ);
    Object[] params = {new Date()};
    Date r = (Date) client.execute("test.testDateTime", params);
    System.out.println(r);
    System.out.println();
  }
  // Binärdaten
  else if (typ.equals("base64")) {
    System.out.println(typ);
    Object[] params = {};
    byte[] r = (byte[]) client.execute("test.testBase64", params);
    OutputStream out = new FileOutputStream("test.gif");
    for (int i = 0; i < r.length; i++) {
      out.write(r[i]);
    }
    out.flush();
    out.close();
    System.out.println();
  }
  // Array
  else if (typ.equals("array")) {
    System.out.println(typ);
    Object[] array = {"Das ist ein String", 4711};
    Object[] params = {array};
    Object[] r = (Object[]) client.execute(
       "test.testArray", params);
    for (Object obj : r) {
      System.out.println(obj);
    }
    System.out.println();
  }
  // Struktur
  else if (typ.equals("struct")) {
    System.out.println(typ);
    Map map = new HashMap();
```

```
      map.put("Vorname", "Hugo");
      map.put("Nachname", "Meier");
      Object[] params = {map};
      Map r = (Map) client.execute("test.testStruct", params);
      System.out.println(r.get("Vorname") + " " + r.get("Nachname"));
      System.out.println();
    }
    // XmlRpcException
    else if (typ.equals("fault")) {
      System.out.println(typ);
      Object[] params = {};
      int r  = (Integer) client.execute("test.testFault", params);
      System.out.println(r);
      System.out.println();
    }
  }
}
```

Für jeden XML-RPC-Datentyp werden nun im Folgenden

- der Parameterwert im XML-Dokument der XML-RPC-Anfrage (<value>-Tag),
- der Rückgabewert im XML-Dokument der XML-RPC-Antwort (<value>-Tag) und
- die Ausgabe des Client

aufgeführt.

Ganzzahl

XML-RPC-Anfrage:

```
<i4>1234</i4>
```

XML-RPC-Antwort:

```
<i4>1234</i4>
```

Ausgabe des Client-Programms:

```
1234
```

Gleitkommazahl

XML-RPC-Anfrage:

```
<double>123.456</double>
```

XML-RPC-Antwort:

```
<double>123.456</double>
```

Ausgabe des Client-Programms:

```
123.456
```

XML-RPC-Anfrage: *Wahrheitswert*

```
<boolean>1</boolean>
```

XML-RPC-Antwort:

```
<boolean>1</boolean>
```

Ausgabe des Client-Programms:

```
true
```

XML-RPC-Anfrage: *Zeichenkette*

```
&lt;-- A &amp; O --&gt;
```

XML-RPC-Antwort:

```
&lt;-- A &amp; O --&gt;
```

Ausgabe des Client-Programms:

```
<-- A & O -->
```

Wenn wie hier kein Datentyp angegeben ist, wird `<string>` unterstellt. Die Zeichen `<`, `>` und `&` haben in XML eine Sonderrolle und werden daher umcodiert.

XML-RPC-Anfrage: *Datum/Zeit*

```
<dateTime.iso8601>20060809T16:38:34</dateTime.iso8601>
```

XML-RPC-Antwort:

```
<dateTime.iso8601>20060809T16:38:34</dateTime.iso8601>
```

Ausgabe des Client-Programms:

```
Wed Aug 09 16:38:34 CEST 2006
```

Die XML-RPC-Anfrage enthält kein `<value>`-Tag. *Binärdaten*

XML-RPC-Antwort:

```
<base64>R0lGODlhNABYAPcAAP...</base64>
```

XML-RPC nutzt das Codierungsverfahren *Base64*, um Binärdaten technisch gesichert in XML-Strukturen zu übertragen. 24 Bit lan-

ge Gruppen der Binärdaten werden in vier Bitfolgen von jeweils 6 Bit zerlegt. Diese 6 Bit werden mit Hilfe der US-ASCII-Zeichen `A – Z`, `a – z`, `0 – 9`, `+`, `/` und `=` codiert (`http://www.ietf.org/rfc/rfc2045.txt`).

Ausgabe des Client-Programms:

Der Client speichert die übertragenen Daten in der Datei *test.gif*.

Array

XML-RPC-Anfrage:

```
<array>
  <data>
    <value>Das ist ein String</value>
    <value><i4>4711</i4></value>
  </data>
</array>
```

XML-RPC-Antwort:

```
<array>
  <data>
    <value>Das ist ein String</value>
    <value><int>4711</int></value>
  </data>
</array>
```

Ausgabe des Client-Programms:

```
Das ist ein String
4711
```

Struktur

XML-RPC-Anfrage:

```
<struct>
  <member>
    <name>Nachname</name>
    <value>Meier</value>
  </member>
  <member>
    <name>Vorname</name>
    <value>Hugo</value>
  </member>
</struct>
```

XML-RPC-Antwort:

```
<struct>
  <member>
    <name>Nachname</name>
    <value>Meier</value>
  </member>
  <member>
    <name>Vorname</name>
```

```
    <value>Hugo</value>
  </member>
</struct>
```

Ausgabe des Client-Programms:

```
Hugo Meier
```

Es wird eine Ausnahme vom Typ `XmlRpcException` ausgelöst, da die Methode `testFault` nicht existiert. *Ausnahme*

Die komplette XML-RPC-Antwort:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodRespose>
  <fault><value><struct>
    <member>
      <name>faultString</name>
      <value>No such handler: test.testFault</value>
    </member>
    <member>
      <name>faultCode</name>
      <value><i4>0</i4></value>
    </member>
  </struct></value></fault>
</methodResponse>
```

Ausgabe:

```
org.apache.xmlrpc.XmlRpcException: No such handler: test.testFault
```

Das Programm `MyReporter` kann genutzt werden, um die zwischen Client und Server ausgetauschten Daten aufzuzeichnen.

Ein *Shutdown hook* sorgt für das ordnungsgemäße Schließen der Log-Dateien, wenn das Programm mit `Strg+C` beendet wird.

MyReporter

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

public class MyReporter extends Thread {
  private static OutputStream logRequest;
  private static OutputStream logResponse;
  private static ServerSocket srv;
  private static int counter;

  private InputStream fromClient;
  private OutputStream toServer;

  public MyReporter(InputStream fromClient, OutputStream toServer) {
    this.fromClient = fromClient;
    this.toServer = toServer;
  }
```

```
public void run() {
  try {
    int c;
    logRequest.write(("#" + counter + "\r\n").getBytes());
    while ((c = fromClient.read()) != -1) {
      logRequest.write(c);
      toServer.write(c);
    }
  }
  catch (IOException e) { }

  try {
    logRequest.write("\r\n\r\n".getBytes());
  }
  catch (IOException e) { }
}

public static void main(String[] args) {
  int port = Integer.parseInt(args[0]);
  String remoteHost = args[1];
  int remotePort = Integer.parseInt(args[2]);
  String file1 = args[3];
  String file2 = args[4];

  // ShutdownHook registrieren
  Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
      try {
        if (logRequest != null) {
          logRequest.flush();
          logRequest.close();
        }
        if (logResponse != null) {
          logResponse.flush();
          logResponse.close();
        }
        if (srv != null)
          srv.close();
      }
      catch (IOException e) { }
    }
  });

  try {
    logRequest = new FileOutputStream(file1);
    logResponse = new FileOutputStream(file2);
    srv = new ServerSocket(port);

    while (true) {
      Socket client = srv.accept();
      InputStream fromClient = client.getInputStream();
      OutputStream toClient = client.getOutputStream();

      Socket socket = new Socket(remoteHost, remotePort);
      InputStream fromServer = socket.getInputStream();
      OutputStream toServer = socket.getOutputStream();
```

```
        counter++;

        // Weiterleitung vom Client an den Server
        (new MyReporter(fromClient, toServer)).start();

        // Weiterleitung vom Server an den Client
        int c;
        logResponse.write(("#" + counter + "\r\n").getBytes());
        while ((c = fromServer.read()) != -1) {
          logResponse.write(c);
          toClient.write(c);
        }
        logResponse.write("\r\n\r\n".getBytes());

        toClient.flush();
        client.close();
        socket.close();
      }
    }
    catch (SocketException e) { }
    catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

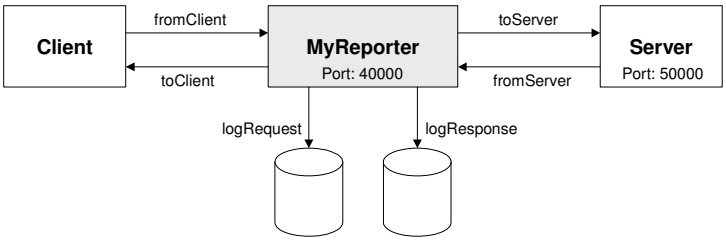


***Bild 6.3**: MyReporter zeichnet HTTP-Anfrage und -Antwort auf*

Compilieren:

```
javac -sourcepath src -cp %XMLRPC_PATH% -d build stc/*.java
```

`MyReporter` starten (Kommando in einer Zeile):

```
start java -cp build MyReporter 40000 localhost 50000
logRequest.txt logResponse.txt
```

logRequest.txt enthält die HTTP-Anfragen und *logResponse.txt* die HTTP-Antworten.

Server starten:

```
start java -cp build;%XMLRPC_PATH% DatentypTestServer 50000
```

Client starten (Kommando in einer Zeile):

```
java -cp build;%XMLRPC_PATH% DatentypTestClient http://localhost:40000
int
...
```

6.3 Komplexe Datenstrukturen

Die Beispiele dieses Kapitels zeigen, wie komplexe Datenstrukturen mit Hilfe einfacher und zusammengesetzter XML-RPC-Typen in ein für XML-RPC geeignetes Format transformiert werden können.

Programm 6.3

Die Handler-Klasse `Warenkorb` verwaltet eine Reihe von Bestellpositionen mit den Attributen *id*, *name*, *preis* und *menge* in einem `Vector`-Objekt und bietet die folgenden Methoden an:

`int addPosition(int id, String name, double preis, int menge)`
fügt eine Position ein.

`Object[] getPositionen()`
liefert alle Positionen des Warenkorbs. Jedes Arrayelement des Rückgabewertes ist selbst wieder ein Array, das als Elemente die Attribute einer Position enthält. Bild 6.4 zeigt die zugehörige XML-Struktur.

Bild 6.4: *Warenkorb als Array von Arrays*

```
<array>
  <data>
    <value>
      <array>
        <data>
          <value><i4>1000</i4></value>
          <value>Hammer</value>
          <value><double>2.5</double></value>
          <value><i4>10</i4></value>
        </data>
      </array>
    </value>
    <value>
      <array>
        <data>
          <value><i4>1010</i4></value>
          <value>Zange</value>
          <value><double>3.99</double></value>
          <value><i4>8</i4></value>
        </data>
      </array>
    </value>
  </data>
</array>
```

Position

```
public class Position {
  private int id;
  private String name;
  private double preis;
  private int menge;

  public Position(int id, String name, double preis, int menge) {
    this.id = id;
    this.name = name;
    this.preis = preis;
    this.menge = menge;
  }

  public Object[] getPosition() {
    Object[] array = new Object[4];
    array[0] = id;
    array[1] = name;
    array[2] = preis;
    array[3] = menge;
    return array;
  }
}
```

Warenkorb

```
import java.util.*;

public class Warenkorb {
  private static Vector<Position> korb = new Vector<Position>();

  public int addPosition(int id, String name, double preis,
    int menge) {

    korb.add(new Position(id, name, preis, menge));
    return 1;
  }

  public Object[] getPositionen() {
    Vector v = new Vector();
    for (Position pos : korb) {
      v.add(pos.getPosition());
    }
    return v.toArray();
  }
}
```

Server

```
import org.apache.xmlrpc.server.*;
import org.apache.xmlrpc.webserver.*;

public class Server {
  public static void main(String[] args) throws Exception {
    int port = Integer.parseInt(args[0]);

    PropertyHandlerMapping phm = new PropertyHandlerMapping();
    phm.addHandler("warenkorb", Warenkorb.class);
```

```
    WebServer webServer = new WebServer(port);
    XmlRpcServer server = webServer.getXmlRpcServer();
    server.setHandlerMapping(phm);
    webServer.start();
  }
}
```

Client

```
import java.util.*;
import java.net.*;
import org.apache.xmlrpc.client.*;

public class Client {
  public static void main(String args[]) throws Exception {
    URL url = new URL(args[0]);
    XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
    config.setServerURL(url);
    XmlRpcClient client = new XmlRpcClient();
    client.setConfig(config);

    Object[] params1 = {1000, "Hammer", 2.5, 10};
    client.execute("warenkorb.addPosition", params1);

    Object[] params2 = {1010, "Zange", 3.99, 8};
    client.execute("warenkorb.addPosition", params2);

    Object[] params3 = {};
    Object[] result = (Object[]) client.execute(
      "warenkorb.getPositionen", params3);
    for (Object obj : result) {
      Object[] array = (Object[]) obj;
      System.out.println("Id: " + (Integer) array[0]);
      System.out.println("Name: " + (String) array[1]);
      System.out.println("Preis: " + (Double) array[2]);
      System.out.println("Menge: " + (Integer) array[3]);
      System.out.println();
    }
  }
}
```

Das Beispiel zeigt auch, dass die Kenntnis der Signatur (mit Rückgabetyp) eines Handlers in der Regel alleine nicht ausreicht, um das Ergebnis des Methodenaufrufs weiterzuverarbeiten. Zusätzlich ist die Datenstruktur (Array von Arrays) und ihre Semantik zu erläutern.

Programm 6.4

Der folgende Dienst ermöglicht die Suche nach Personen, die auf der Serverseite in einer Datenbank gespeichert sind. Das Suchergebnis ist eine Liste von Personen mit den zugehörigen Adressen.

Personen und Adressen sind in einer MySQL-Datenbank in zwei miteinander verknüpften Tabellen gespeichert.

```
create table person (
  persid integer not null auto_increment,
  name varchar(30),
  vorname varchar(30),
  gebdatum date,
  primary key (persid)
)

create table adresse (
  adressid integer not null auto_increment,
  persid integer not null,
  plz char(5),
  ort varchar(30),
  primary key (adressid),
  foreign key (persid) references person (persid)
)
```

Die Klasse `PersonQuery` enthält die Methode

```
ArrayList<Person> getPersonen(String name),
```

die mit Hilfe des Suchbegriffs `name`, der auch so genannte Wildcard-Zeichen wie `%` und _ enthalten darf, in der Datenbank nach Personen, deren Namen dem Kriterium entsprechen, sucht und eine Liste von `Person`-Objekten zurückliefert. Ein `Person`-Objekt enthält eine Liste von möglicherweise mehreren `Adresse`-Objekten.

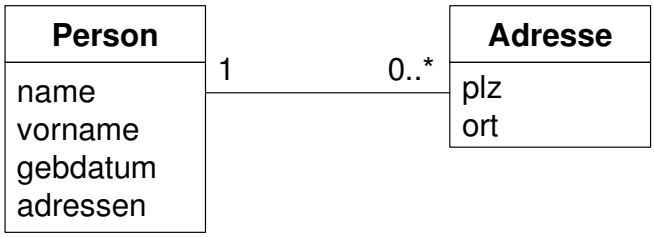


***Bild 6.5**: Die Person-Adresse-Beziehung*

Person

```
import java.util.*;
import java.text.*;

public class Person {
  private String name;
  private String vorname;
  private Date gebdatum;
  private ArrayList<Adresse> adressen;
```

```
  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public void setVorname(String vorname) {
      this.vorname = vorname;
  }

  public String getVorname() {
    return vorname;
  }

  public void setGebdatum(Date gebdatum) {
    this.gebdatum = gebdatum;
  }

  public Date getGebdatum() {
    return gebdatum;
  }

  public void setAdressen(ArrayList<Adresse> adressen) {
    this.adressen = adressen;
  }

  public ArrayList<Adresse> getAdressen() {
    return adressen;
  }

  public void print() {
    System.out.println("Name: " + name);
    System.out.println("Vorname: " + vorname);

    SimpleDateFormat f = new SimpleDateFormat("dd.MM.yyyy");
    if (gebdatum != null)
      System.out.println("Gebdatum: " + f.format(gebdatum));
    else
      System.out.println("Gebdatum: null");

    if (adressen != null) {
      for (Adresse adr : adressen) {
        adr.print();
      }
    }
  }
}
```

Adresse

```
public class Adresse {
  private String plz;
  private String ort;

  public void setPlz(String plz) {
    this.plz = plz;
  }
```

```
  public String getPlz() {
    return plz;
  }

  public void setOrt(String ort) {
    this.ort = ort;
  }

  public String getOrt() {
    return ort;
  }

  public void print() {
    System.out.println("PLZ :" + plz);
    System.out.println("Ort :" + ort);
  }
}
```

PersonQuery

```
import java.io.*;
import java.util.*;
import java.sql.*;

public class PersonQuery {
  private String url;
  private String user;
  private String password;

  public PersonQuery() {
    try {
      InputStream input = this.getClass().getResourceAsStream(
        "dbconnect.properties");
      Properties prop = new Properties();
      prop.load(input);
      input.close();
      String driver = prop.getProperty("driver");
      url = prop.getProperty("url");
      user = prop.getProperty("user");
      password = prop.getProperty("password");
      Class.forName(driver);
    }
    catch (Exception e) {
      System.err.println(e);
    }
  }

  public ArrayList<Person> getPersonen(String name) {
    ArrayList<Person> personen = new ArrayList<Person>();
    Connection con = null;

    try {
      con = DriverManager.getConnection(url, user, password);
      String sql = "select persid, name, vorname, gebdatum " +
        "from person where name like ? order by name, vorname";
      PreparedStatement pstmt = con.prepareStatement(sql);
      pstmt.setString(1, name);
```

```
    ResultSet rs = pstmt.executeQuery();

    while (rs.next()) {
      Person p = new Person();
      int persid = rs.getInt(1);
      p.setName(rs.getString(2));
      p.setVorname(rs.getString(3));
      p.setGebdatum(rs.getDate(4));
      p.setAdressen(getAdressen(persid, con));
      personen.add(p);
    }

    rs.close();
    pstmt.close();
  }
  catch (SQLException e) {
    System.err.println(e);
  }
  finally {
    try {
      if (con != null)
        con.close();
    }
    catch (SQLException e) { }
    return personen;
  }
}

private ArrayList<Adresse> getAdressen(int persid, Connection con) {
  ArrayList<Adresse> adressen = new ArrayList<Adresse>();

  try {
    String sql = "select plz, ort from adresse " +
      "where persid = ? order by plz";
    PreparedStatement pstmt = con.prepareStatement(sql);
    pstmt.setInt(1, persid);
    ResultSet rs = pstmt.executeQuery();
    while (rs.next()) {
      Adresse a = new Adresse();
      a.setPlz(rs.getString(1));
      a.setOrt(rs.getString(2));
      adressen.add(a);
    }

    rs.close();
    pstmt.close();
  }
  catch (Exception e) {
    System.err.println(e);
  }
  finally {
    return adressen;
  }
}
```

```
  // Testprogramm
  public static void main(String[] args) throws Exception {
    PersonQuery query = new PersonQuery();
    ArrayList<Person> personen = query.getPersonen("%");
    for (Person pers : personen) {
      pers.print();
      System.out.println();
    }
  }
}
```

Die DB-Verbindungsparameter befinden sich in der Datei *dbconnect.properties*. Die Methode `getPersonen` kann unabhängig von XML-RPC getestet werden (`main`).

Um das Suchergebnis mittels XML-RPC zum Client zu übertragen, muss das Modell aus Bild 6.5 in eine für XML-RPC geeignete Struktur transformiert werden. Diese darf nur die Datentypen aus Kapitel 6.2 enthalten.

Diesem Zweck dienen die beiden Hilfsklassen `PersonHelper` und `AdresseHelper`. Sie enthalten jeweils zwei statische Methoden: eine, die ein `Adresse`- bzw. `Person`-Objekt in eine *Map* transformiert, und eine, die aus einer *Map* wiederum ein `Adresse`- bzw. `Person`-Objekt erzeugt. Erstere wird auf der Serverseite, letztere auf der Clientseite benötigt.

AdresseHelper

```
import java.util.*;

public class AdresseHelper {
  private final static String PLZ = "plz";
  private final static String ORT = "ort";

  public static Map toMap(Adresse adr) {
    Map map = new HashMap();

    String plz = adr.getPlz();
    if (plz != null)
      map.put(PLZ, plz);

    String ort = adr.getOrt();
    if (ort != null)
      map.put(ORT, ort);

    return map;
  }

  public static Adresse fromMap(Map map) {
    Adresse adr = new Adresse();

    Object obj = map.get(PLZ);
```

```
    if (obj != null)
      adr.setPlz((String) obj);

    obj = map.get(ORT);
    if (obj != null)
      adr.setOrt((String) obj);

    return adr;
  }
}
```

PersonHelper

```
import java.util.*;

public class PersonHelper {
  private final static String NAME = "name";
  private final static String VORNAME = "vorname";
  private final static String GEBDATUM = "gebdatum";
  private final static String ADRESSEN = "adressen";

  public static Map toMap(Person pers) {
    Map map = new HashMap();

    String name = pers.getName();
    if (name != null)
      map.put(NAME, name);

    String vorname = pers.getVorname();
    if (vorname != null)
      map.put(VORNAME, vorname);

    Date gebdatum = pers.getGebdatum();
    if (gebdatum != null)
      map.put(GEBDATUM, gebdatum);

    ArrayList<Adresse> adressen = pers.getAdressen();
    if (adressen != null) {
      Object[] array = new Object[adressen.size()];
      int i = 0;
      for (Adresse adr : adressen) {
        array[i++] = AdresseHelper.toMap(adr);
      }
      map.put(ADRESSEN, array);
    }

    return map;
  }

  public static Person fromMap(Map map) {
    Person pers = new Person();

    Object obj = map.get(NAME);
    if (obj != null)
      pers.setName((String) obj);
```

```
    obj = map.get(VORNAME);
    if (obj != null)
      pers.setVorname((String) obj);

    obj = map.get(GEBDATUM);
    if (obj != null)
      pers.setGebdatum((Date) obj);

    obj = map.get(ADRESSEN);
    if (obj != null) {
      ArrayList<Adresse> adressen = new ArrayList<Adresse>();
      for (Object o : (Object[]) obj) {
        adressen.add(AdresseHelper.fromMap((Map) o));
      }
      pers.setAdressen(adressen);
    }

    return pers;
  }
}
```

Der Handler `getPersonen` der Klasse `PersonQueryHandler` nutzt die `PersonQuery`-Methode `getPersonen` und transformiert die `Person`-Objekte. Rückgabewert ist ein Array von Maps. Eine jede Map entspricht einer gefundenen Person. Bild 6.6 zeigt das Ergebnis einer Transformation.

PersonQuery-Handler

```
import java.util.*;

public class PersonQueryHandler {
  private static PersonQuery query = new PersonQuery();

  public Object[] getPersonen(String name) {
    ArrayList<Person> personen = query.getPersonen(name);
    Object[] array = new Object[personen.size()];
    if (personen != null) {
      int i = 0;
      for (Person pers : personen) {
        array[i++] = PersonHelper.toMap(pers);
      }
    }
    return array;
  }
}
```

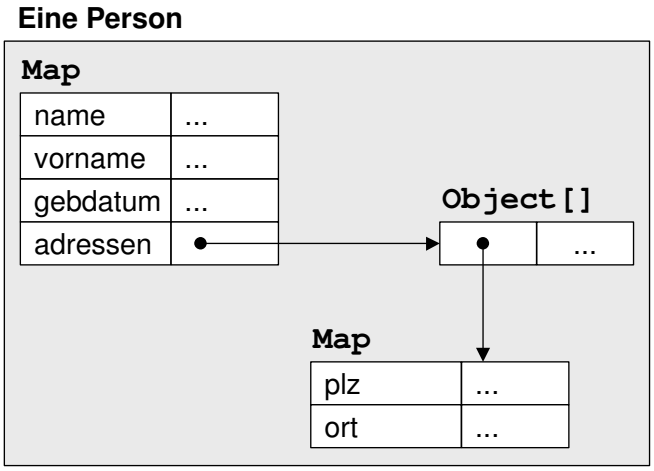


***Bild 6.6**: Abbildung der Person-Adresse-Beziehung auf Map und Array*

Server

```
import org.apache.xmlrpc.server.*;
import org.apache.xmlrpc.webserver.*;

public class Server {
  public static void main(String[] args) throws Exception {
    int port = Integer.parseInt(args[0]);

    PropertyHandlerMapping phm = new PropertyHandlerMapping();
    phm.addHandler("personQuery", PersonQueryHandler.class);

    WebServer webServer = new WebServer(port);
    XmlRpcServer server = webServer.getXmlRpcServer();
    server.setHandlerMapping(phm);
    webServer.start();
  }
}
```

Client

```
import java.util.*;
import java.net.*;
import org.apache.xmlrpc.client.*;

public class Client {
  public static void main(String args[]) throws Exception {
    URL url = new URL(args[0]);
    String name = args[1];

    XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
    config.setServerURL(url);
    XmlRpcClient client = new XmlRpcClient();
    client.setConfig(config);

    Object[] params = {name};
    Object result = client.execute("personQuery.getPersonen", params);
```

```
    if (result != null) {
      for (Object obj : (Object[]) result) {
        Person pers = PersonHelper.fromMap((Map) obj);
        pers.print();
        System.out.println();
      }
    }
  }
}
```

Hier wird jede Map des von `execute` gelieferten Arrays in ein `Person`-Objekt transformiert.

6.4 Dynamische Proxies

Mit Hilfe so genannter *dynamischer Proxies* kann die Programmierung des Clients vereinfacht werden. Ein *dynamischer Proxy* ist eine Klasse, die nicht vom Compiler, sondern erst zur Laufzeit dynamisch generiert wird.

Ziel ist es, die entfernte, vom Server implementierte Methode wie eine "normale" lokale Methode auf der Clientseite aufzurufen.

Dazu muss zunächst die Handler-Klasse ein *Interface* implementieren. Dasselbe Interface wird auch vom Client verwendet.

Wir erläutern die Vorgehensweise anhand des Echo-Dienstes aus Programm 6.1.

Programm 6.5

Interface Echo

```
public interface Echo {
  String getEcho(String s);
  String getEchoWithDate(String s);
}
```

Implementierung EchoImpl

```
import java.util.*;
import java.text.*;

public class EchoImpl implements Echo {
  public String getEcho(String s) {
    return s;
  }

  public String getEchoWithDate(String s) {
    SimpleDateFormat f = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
    return "[" + f.format(new Date()) + "] " + s;
  }
}
```

Der beim Aufruf der Methode `addHandler` benutzte Key für die Handler-Klasse muss mit dem voll qualifizierten Namen des *Interfaces* (`Paket.Interface`) übereinstimmen.

EchoServer

```
import org.apache.xmlrpc.server.*;
import org.apache.xmlrpc.webserver.*;

public class EchoServer {
  public static void main(String[] args) throws Exception {
    int port = Integer.parseInt(args[0]);

    PropertyHandlerMapping phm = new PropertyHandlerMapping();
    phm.addHandler("Echo", EchoImpl.class);

    WebServer webServer = new WebServer(port);
    XmlRpcServer server = webServer.getXmlRpcServer();
    server.setHandlerMapping(phm);
    webServer.start();
  }
}
```

Auf der Clientseite wird nun nicht das `XmlRpcClient`-Objekt `client` direkt zum Aufruf der entfernten Methode benutzt, sondern zunächst eine Instanz der Klasse

```
org.apache.xmlrpc.client.util.ClientFactory
```

wie folgt erzeugt:

```
ClientFactory factory = new ClientFactory(client);
```

Die `ClientFactory`-Methode

```
Object newInstance(Class c)
```

wird mit dem Interface `Echo.class` als Argument aufgerufen. Zurückgeliefert wird eine Implementierung dieses Interfaces, die intern `client` nutzt, um die entfernte Methode aufzurufen:

```
Echo echo = (Echo) factory.newInstance(Echo.class);
```

Nun können die `Echo`-Methoden aufgerufen werden, z.B.

```
String s = echo.getEcho("Hallo");
```

EchoClient

```
import java.net.*;
import org.apache.xmlrpc.client.*;
import org.apache.xmlrpc.client.util.ClientFactory;

public class EchoClient {
  public static void main(String args[]) throws Exception {
    URL url = new URL(args[0]);
    XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
    config.setServerURL(url);
    XmlRpcClient client = new XmlRpcClient();
```

```
    client.setConfig(config);

    ClientFactory factory = new ClientFactory(client);
    Echo echo = (Echo) factory.newInstance(Echo.class);

    String s = echo.getEcho("Hallo");
    System.out.println(s);

    String t = echo.getEchoWithDate("Hallo");
    System.out.println(t);
  }
}
```

6.5 Filterung von IP-Adressen

Mit den folgenden `WebServer`-Methoden kann der Webserver nur bestimmten Clients den Methodenaufruf gestatten bzw. verbieten.

`void` **`setParanoid`**`(boolean p)`
schaltet die Filterung von IP-Adressen ein (`true`) bzw. aus (`false`). Der Webserver akzeptiert standardmäßig Anfragen aller IP-Adressen. Mit `true` wird zunächst keine Client-Verbindung akzeptiert.

`void` **`acceptClient`**`(String address)`
fügt die IP-Adresse `address` in die *Liste der akzeptierten IP- Adressen* ein. `address` kann `*` enthalten, um einen Nummernkreis anzugeben (z.B. `194.94.124.*`). `acceptClient` wirkt nur, wenn `setParanoid(true)` aufgerufen wurde.

`void` **`denyClient`**`(String address)`
fügt die IP-Adresse `address` in die *Liste der ausgeschlossenen IP-Adressen* ein. `address` kann `*` enthalten, um einen Nummernkreis anzugeben (z.B. `194.94.124.*`). `denyClient` wirkt nur, wenn `setParanoid(true)` aufgerufen wurde.

Das folgende Beispiel zeigt den Einsatz der IP-Filterung. Die Liste der akzeptierten (`accept`) bzw. ausgeschlossenen (`deny`) IP-Adressen wird der Property-Datei *ip.txt* entnommen. ***Programm 6.6***

Beispiel:

```
accept: 127.0.0.1 10.108.105.* 10.108.1.51
```

EchoServer

```
import org.apache.xmlrpc.server.*;
import org.apache.xmlrpc.webserver.*;
import java.util.*;
import java.io.*;

public class EchoServer {
  public static void main(String[] args) throws Exception {
    int port = Integer.parseInt(args[0]);

    PropertyHandlerMapping phm = new PropertyHandlerMapping();
    phm.addHandler("echo", Echo.class);

    WebServer webServer = new WebServer(port);
    webServer.setParanoid(true);

    FileInputStream in = new FileInputStream("ip.txt");
    Properties ip = new Properties();
    ip.load(in);
    in.close();

    String addr = ip.getProperty("accept");
    if (addr != null) {
      StringTokenizer st = new StringTokenizer(addr, " ");
      while (st.hasMoreTokens())
        webServer.acceptClient(st.nextToken());
    }

    addr = ip.getProperty("deny");
    if (addr != null) {
      StringTokenizer st = new StringTokenizer(addr, " ");
      while (st.hasMoreTokens())
        webServer.denyClient(st.nextToken());
    }

    XmlRpcServer server = webServer.getXmlRpcServer();
    server.setHandlerMapping(phm);
    webServer.start();
  }
}
```

Das Client-Programm löst eine Ausnahme aus, wenn die IP-Adresse des Client-Rechners nicht zugelassen ist.

6.6 Einbettung von XML-RPC in Apache Tomcat

Bisher wurde in allen Beispielen die Klasse `WebServer` benutzt. Sie implementiert einen einfachen HTTP-Server, der zudem XML-RPC-Anfragen verarbeiten kann. Bisweilen ist es aus Performance- und Sicherheitsgründen erforderlich, die XML-RPC-Verarbeitung in einen anderen, marktgängigen Webserver zu integrieren.

Wir zeigen diese Integration am Beispiel von *Apache Tomcat*, der einen *Servlet-Container* als Ablaufumgebung für Webanwendungen bietet. Im Folgenden setzen wir den Umgang mit *Tomcat* und Grundlagen zur *Servlet-Technologie* voraus.

Programm 6.7

Die Klasse

```
org.apache.xmlrpc.webserver.XmlRpcServletServer
```

ist Subklasse von `XmlRpcServer` und besitzt die folgende Methode:

```
void execute(javax.servlet.http.HttpServletRequest,
    javax.servlet.http.HttpServletResponse)
    throws javax.servlet.ServletException, java.io.IOException
```

Sie verarbeitet die XML-RPC-Anfrage und liefert die XML-RPC-Antwort zurück.

Diese Methode kann nun sehr einfach in der `doPost`-Methode eines Servlets genutzt werden.

Die folgende abstrakte Klasse `xmlrpc.AbstractXmlRpcServlet` dient als universelle Basisklasse für eigene Servlets.

AbstractXmlRpc-Servlet

```
package xmlrpc;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.xmlrpc.*;
import org.apache.xmlrpc.webserver.*;
import org.apache.xmlrpc.server.*;

public abstract class AbstractXmlRpcServlet extends HttpServlet {
  private XmlRpcServletServer server;

  public final void init(ServletConfig config)
    throws ServletException {

    super.init(config);
    try {
      PropertyHandlerMapping phm = new PropertyHandlerMapping();
      addHandlers(phm);
      server = new XmlRpcServletServer();
      server.setHandlerMapping(phm);
    }
    catch (XmlRpcException e) {
      throw new ServletException(e);
    }
  }
```

```
  public abstract void addHandlers(PropertyHandlerMapping phm)
    throws XmlRpcException;

  public final void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {

    server.execute(request, response);
  }
}
```

Zur Demonstration ist die Klasse `echo.EchoServlet` von dieser abstrakten Klasse abgeleitet. Sie implementiert die Methode `addHandlers`. Somit haben Subklassen nur noch die Aufgabe, die geforderten Handler zu registrieren.

EchoServlet

```
package echo;

import org.apache.xmlrpc.*;
import org.apache.xmlrpc.server.*;
import xmlrpc.AbstractXmlRpcServlet;

public class EchoServlet extends AbstractXmlRpcServlet {
  public void addHandlers(PropertyHandlerMapping phm)
    throws XmlRpcException {

    phm.addHandler("echo", Echo.class);
  }
}
```

Die hier verwendete Klasse `echo.Echo` entspricht der in Programm 6.1 verwendeten Klasse. Der einzige Unterschied ist, dass eine `package`-Klausel hinzugekommen ist.

Die übersetzten Klassen `AbstractXmlRpcServlet`, `EchoServlet` und `Echo` gehören gemäß ihrer Paketstruktur in das Verzeichnis *classes* unter *WEB-INF*. Die für XML-RPC erforderlichen JAR-Dateien müssen in das Verzeichnis *lib* unter *WEB-INF* kopiert werden.

Die Deskriptordatei *web.xml* hat den folgenden Inhalt:

web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
  version="2.4">

  <servlet>
    <servlet-name>EchoServlet</servlet-name>
    <servlet-class>echo.EchoServlet</servlet-class>
  </servlet>
```

```
  <servlet-mapping>
    <servlet-name>EchoServlet</servlet-name>
    <url-pattern>/servlet/echo</url-pattern>
  </servlet-mapping>
</web-app>
```

Zum Schluss muss diese Webanwendung Tomcat bekannt gemacht werden. Hierzu muss die Datei *xmlrpc.xml* mit dem `<Context>`-Tag (*path="/xmlrpc"*) in das Verzeichnis

```
<CATALINA_HOME>\conf\Catalina\localhost
```

kopiert werden.

Nun kann Tomcat gestartet werden.

Der Client `EchoClient` entspricht der gleichnamigen Klasse in Programm 6.1.

Basic Authentication

Apache XML-RPC unterstützt auch die Authentifizierung mit Benutzername und Passwort. XML-RPC nutzt das HTTP-Verfahren *Basic Authentication* (`http://www.ietf.org/rfc/rfc2617.txt`).

Die Klasse `XmlRpcClientConfigImpl` bietet hierzu die Methoden

`void` **`setBasicUserName`**`(String user)` und
`void` **`setBasicPassword`**`(String password)`.

`user` und `password` werden nach dem *Base64-Verfahren* codiert (siehe Kapitel 6.2) und in einem *Basic Authentication Header* per HTTP an den Server geschickt. Zu beachten ist, dass es sich hierbei um keine echte Verschlüsselung von Benutzername und Passwort mit dem Ziel der Geheimhaltung handelt.

Beispiel: Ist der User "hugo" und das Passwort "oguh", so lautet der HTTP-Header:

```
Authorization: Basic aHVnbzpvZ3Vo
```

"`aHVnbzpvZ3Vo`" wird nach dem Base64-Verfahren auf der Serverseite zu "`hugo:oguh`" decodiert.

Programm 6.8

EchoClient

```
import java.net.*;
import org.apache.xmlrpc.client.*;

public class EchoClient {
  public static void main(String args[]) throws Exception {
    URL url = new URL(args[0]);
    String user = args[1];
    String password = args[2];

    XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
    config.setServerURL(url);
    config.setBasicUserName(user);
    config.setBasicPassword(password);
```

```
    XmlRpcClient client = new XmlRpcClient();
    client.setConfig(config);

    Object[] params = {"Hallo"};
    String s = (String) client.execute("echo.getEcho", params);
    System.out.println(s);

    String t = (String) client.execute(
      "echo.getEchoWithDate", params);
    System.out.println(t);
  }
}
```

Tomcat muss für die Authentifizierung konfiguriert werden. Die Deskriptordatei *web.xml* ist wie folgt zu ergänzen:

web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
  version="2.4">

  <servlet>
    <servlet-name>EchoServlet</servlet-name>
    <servlet-class>echo.EchoServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>EchoServlet</servlet-name>
    <url-pattern>/servlet/echo</url-pattern>
  </servlet-mapping>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>echo</web-resource-name>
      <url-pattern>/servlet/echo</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>user</role-name>
    </auth-constraint>
  </security-constraint>

  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Test</realm-name>
  </login-config>

  <security-role>
    <role-name>user</role-name>
  </security-role>
</web-app>
```

Die zugriffsberechtigten User werden der Einfachheit halber in die XML-Datei

```
<CATALINA_HOME>\conf\tomcat-users.xml
```

eingetragen:

```
<tomcat-users>
  <role rolename="user"/>
  <user username="hugo" password="oguh" roles="user"/>
</tomcat-users>
```

SSL

Zusätzlich kann eine Verschlüsselung mit Server-Authentifizierung genutzt werden. *Secure Socket Layer* (SSL) ist ein weit verbreitetes Verfahren, das auf Public-Key-Verschlüsselung basiert.

Programm 6.9

Zur Nutzung von SSL müssen Server und Client vorbereitet werden. Der Quellcode der Programme ist der gleiche wie in Programm 6.8.

Konfiguration des Servers

Zunächst wird ein Schlüsselpaar (private key, public key) mit einem "self-signed" Zertifikat erzeugt. Dies reicht für Testzwecke aus. Anschließend wird das Zertifikat (mit dem öffentlichen Schlüssel des Servers) exportiert.

createKeystore

Wir benutzen hierzu das JDK-Tool *keytool*.

```
keytool -genkey
-dname "CN=localhost, OU=FB 08, O=Hochschule Niederrhein, C=DE"
-alias tomcat -keyalg RSA -validity 60 -keystore keystore
-keypass tomcat -storepass tomcat
```

```
keytool -export -rfc -alias tomcat -file tomcat.cer
-keystore keystore -storepass tomcat
```

(Die beiden Kommandos sind im DOS-Fenster jeweils in einer Zeile einzugeben.)

server.xml

In `<CATALINA_HOME>\conf\server.xml` wird nun ein Konnektor für den Port 8443 eingerichtet:

```
<Connector port="8443" maxHttpHeaderSize="8192"
  maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
  enableLookups="false" disableUploadTimeout="true"
  acceptCount="100" scheme="https" secure="true"
  clientAuth="false" sslProtocol="TLS"
  keystoreFile="<Verzeichnis>/keystore"
  keystorePass="tomcat" />
```

Konfiguration des Client

Auf der Clientseite wird das Zertifikat in den Speicher *truststore* importiert.

createTruststore

```
keytool -import -noprompt -alias tomcat -file tomcat.cer
-keystore truststore -storepass catalina
```

(Das Kommando ist im DOS-Fenster in einer Zeile einzugeben.)

Aufruf des Client

Angaben zum Speicherort des Zertifikats werden beim Aufruf des Client-Programms als Systemeigenschaften mitgegeben:

```
java -Djavax.net.ssl.trustStore=truststore
-Djavax.net.ssl.trustStorePassword=catalina
-cp build;%XMLRPC_PATH% EchoClient
https://localhost:8443/xmlrpc/servlet/echo hugo oguh
```

(Das Kommando ist im DOS-Fenster in einer Zeile einzugeben.)

Zu beachten ist, dass statt `http` hier `https` (HTTP over SSL) anzugeben ist.

6.7 Nutzung einer Erweiterung in Apache XML-RPC

In diesem Kapitel gehen wir auf die in Kapitel 6.1 angesprochene Erweiterung in Apache XML-RPC 3.0 ein. Dazu muss auf beiden Seiten (Client und Server) Apache XML-RPC zum Einsatz kommen.

Da Apache XML-RPC in der Voreinstellung die offizielle XML-RPC-Spezifikation erfüllt, müssen für die Nutzung der nicht dem Standard entsprechenden Erweiterung sowohl der Client als auch der Server besonders konfiguriert werden. Hierzu ist die Eigenschaft `enabledForExtensions` zu setzen.

Programm 6.10

Programm 6.10 zeigt, dass Klassen, die das Interface `java.io.Serializable` implementieren, als Datentypen für XML-RPC-Handler verwendet werden können. Zudem nutzen wir die Programmiertechnik aus Kapitel 6.4 ("Dynamische Proxies"). Die fachliche Aufgabenstellung wurde aus Programm 6.3 übernommen.

Klasse Position

```
public class Position implements java.io.Serializable {
  private int id;
  private String name;
  private double preis;
  private int menge;

  public Position() { }
  public Position(int id, String name, double preis, int menge) {
    this.id = id;
```

```
    this.name = name;
    this.preis = preis;
    this.menge = menge;
  }

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public double getPreis() {
    return preis;
  }

  public void setPreis(double preis) {
    this.preis = preis;
  }

  public int getMenge() {
    return menge;
  }

  public void setMenge(int menge) {
    this.menge = menge;
  }
}
```

Interface Warenkorb

```
import java.util.*;

public interface Warenkorb {
  int addPosition(Position pos);
  Vector<Position> getPositionen();
}
```

Implementierung WarenkorbImpl

```
import java.util.*;

public class WarenkorbImpl implements Warenkorb {
  private static Vector<Position> korb = new Vector<Position>();

  public int addPosition(Position pos) {
    korb.add(pos);
    return 1;
  }
```

```
  public Vector<Position> getPositionen() {
    return korb;
  }
}
```

Mit Hilfe der Klasse

`org.apache.xmlrpc.server.XmlRpcServerConfigImpl`

kann ein `XmlRpcServer`-Objekt konfiguriert werden.

Die `XmlRpcServerConfigImpl`-Methode

`void` **`setEnabledForExtensions`**`(boolean ext)`

aktiviert (`true`) bzw. deaktiviert (`false`) die Erweiterungen.

Mit dem Aufruf der `XmlRpcServer`-Methode `setConfig` wird die Konfiguration `config` für den Server gesetzt: `setConfig(config)`.

Server

```
import org.apache.xmlrpc.server.*;
import org.apache.xmlrpc.webserver.*;

public class Server {
  public static void main(String[] args) throws Exception {
    int port = Integer.parseInt(args[0]);

    PropertyHandlerMapping phm = new PropertyHandlerMapping();
    phm.addHandler("Warenkorb", WarenkorbImpl.class);

    WebServer webServer = new WebServer(port);
    XmlRpcServer server = webServer.getXmlRpcServer();
    XmlRpcServerConfigImpl config = new XmlRpcServerConfigImpl();
    config.setEnabledForExtensions(true);
    server.setConfig(config);
    server.setHandlerMapping(phm);
    webServer.start();
  }
}
```

Die `XmlRpcClientConfigImpl`-Methode

`void` **`setEnabledForExtensions`**`(boolean ext)`

aktiviert (`true`) bzw. deaktiviert (`false`) die Erweiterungen.

Client

```
import java.util.*;
import java.net.*;
import org.apache.xmlrpc.client.*;
import org.apache.xmlrpc.client.util.ClientFactory;

public class Client {
  public static void main(String args[]) throws Exception {
    URL url = new URL(args[0]);
    XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
```

```
    config.setServerURL(url);
    config.setEnabledForExtensions(true);
    XmlRpcClient client = new XmlRpcClient();
    client.setConfig(config);

    ClientFactory factory = new ClientFactory(client);
    Warenkorb korb = (Warenkorb) factory.newInstance(Warenkorb.class);

    Position pos1 = new Position(1000, "Hammer", 2.5, 10);
    korb.addPosition(pos1);

    Position pos2 = new Position(1010, "Zange", 3.99, 8);
    korb.addPosition(pos2);

    Vector<Position> v = korb.getPositionen();
    for (Position pos : v) {
      System.out.println("Id: " + pos.getId());
      System.out.println("Name: " + pos.getName());
      System.out.println("Preis: " + pos.getPreis());
      System.out.println("Menge: " + pos.getMenge());
      System.out.println();
    }
  }
}
```

6.8 Aufgaben

1. Entwickeln Sie einen XML-RPC-Server, der an ihn übermittelte Nachrichten (Username, Text) in einer Datei speichert.

2. Entwickeln Sie einen XML-RPC-Server, der Adressen in einer Datenbank verwaltet:

   ```
   id nachname  vorname   telefon          email
    1 Meier     Hugo      02102/112233     h.meier@xyz.de
    ...
   ```

   Der Client kann

   - einen Eintrag erfassen:

     ```
     boolean add(String nachname, String vorname, String telefon,
       String email)
     ```

   - einen Eintrag löschen:

     ```
     boolean remove(int id)
     ```

   - alle Einträge auflisten:

     ```
     Object[] getAll()
     ```

   - Einträge suchen:

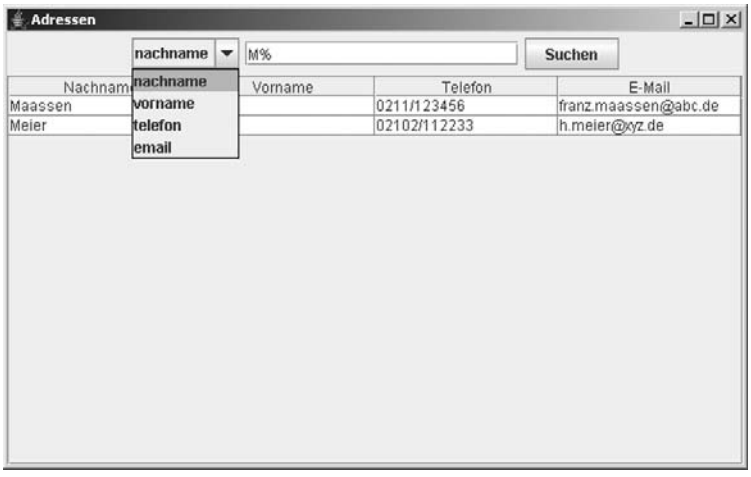
     ```
     Object[] lookup(String spalte, String muster)
     ```

Die einzelnen Funktionen sollen parametergesteuert (Aufrufparameter) abgerufen werden können:

Funktion	*Parameter*
`getAll`	`<url>`
`remove`	`<url> <id>`
`lookup`	`<url> <spalte> <muster>`
`add`	`<url> <nachname> <vorname> <telefon> <email>`

3. Entwickeln Sie als Ergänzung zur Aufgabe 2 einen Client mit grafischer Oberfläche, mit dem nach Adressen gesucht werden kann (*lookup*).

***Bild 6.7**: Adressen suchen*



4. Ein XML-RPC-Server soll Eintritts- und Austrittszeiten in einer Datenbank verwalten:

```
id      beginn                ende
hugo    2006-05-13 08:01:43 2006-05-13 12:15:43
hugo    2006-05-13 13:10:12 2006-05-13 15:45:02
hugo    2006-05-13 16:25:11
...
```

Der Client kann

- Beginn und Ende zu einer ID erfassen:

  `String setTime(String id)`

- die Gesamtzeit zu einer ID abfragen:

  `String getTotalTime(String id)`

5. Diese Aufgabe setzt *PHP-Grundkenntnisse* voraus. Für die vier in Aufgabe 2 implementierten XML-RPC-Dienste `add`, `remove`, `getAll` und `lookup` soll eine Web-Oberfläche mit Hilfe von PHP realisiert werden. Die einzelnen PHP-Skripte implementieren also XML-RPC-Clients und generieren HTML-Seiten zur Anzeige der Ergebnisse.

   Zur Lösung dieser Aufgabe kann beispielsweise die XML-RPC-Library für PHP unter der Adresse

   ```
   http://sourceforge.net/projects/phpxmlrpc/
   ```

   heruntergeladen werden. Die Include-Datei *xmlrpc.inc* enthält den Code für die Client-Funktionalität.

7 Entfernter Methodenaufruf mit RMI

In diesem Kapitel stellen wir ein System vor, das die Kommunikation zwischen Objekten, die auf verschiedenen Rechnern erzeugt sind, ermöglicht. Im Gegensatz zur XML-RPC-Spezifikation aus Kapitel 6 sind die Datentypen der Methodenaufrufparameter nicht eingeschränkt. Neben (fast) beliebigen Objekten kann auch das Verhalten (Bytecode) übertragen werden, so dass flexible Anwendungen realisiert werden können. Allerdings muss sowohl der Client als auch der Server in Java programmiert sein.

7.1 Remote Method Invocation

Das Protokoll *Remote Method Invocation* (RMI) setzt auf TCP/IP auf und verbirgt die Details einer Netzverbindung. RMI hat folgende Eigenschaften:

Eigenschaften von RMI

- Mit RMI können Methoden für Objekte aufgerufen werden, die von einer anderen virtuellen Maschine (JVM) erzeugt und verwaltet werden – in der Regel auf einem anderen Rechner.
- Für den Entwickler sieht der entfernte Methodenaufruf wie ein ganz normaler lokaler Aufruf aus.
- Entfernt aufrufbare Methoden werden in einem *Interface* deklariert. Nur hierüber kann der Client mit einem entfernten Objekt kommunizieren. Das Interface stellt einen so genannten *Vertrag* zwischen Client und Server dar.
- *Netzspezifischer Code*, der die Codierung und Übertragung von Aufrufparametern und Rückgabewerten ermöglicht wird ab Java SE 5.0 dynamisch zur Laufzeit generiert.
- Um für den ersten Aufruf einer entfernten Methode eine "Referenz" auf das entfernte Objekt, das diese Methode anbietet, zu erhalten, kann der Client einen so genannten *Namensdienst* (Registry) nutzen.
- RMI bietet Mechanismen sowohl für die Übertragung von Objekten als auch für die Übertragung und das Laden des Bytecodes der zugehörigen Klassen, falls diese lokal nicht vorhanden sind.
- RMI ist eine rein Java-basierte Lösung, d.h. Client und Server müssen in Java programmiert sein.

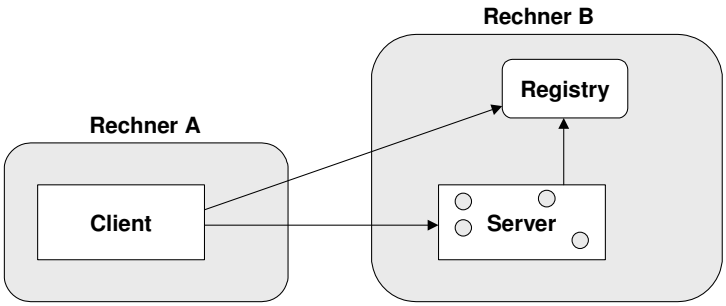
Begriffserklärung Namensdienst

Mit Hilfe eines *Namensdienstes* können Dienste zentral veröffentlicht werden, sodass Clients diese über ein Netz finden und nutzen können. Namensdienste ordnen den Adressen von Ressourcen (beispielsweise entfernten Objekten) eindeutige Namen zu.

Die Adresse einer Ressource enthält alle Informationen, die ein Client braucht, um mit der Ressource zu kommunizieren. Um eine "Referenz" auf die gewünschte Ressource zu erhalten, übergibt der Client dem Namensdienst den Namen, unter dem die Ressource angemeldet ist. Als Ergebnis erhält er die Referenz, mit der nun eine Verbindung zur Ressource aufgebaut werden kann.

Bild 7.1 zeigt den allgemeinen Fall einer mit RMI realisierten Client-Server-Anwendung.

***Bild 7.1**: RMI-Anwendung*



Der Client erhält über die Registry eine "Referenz" auf ein entferntes Objekt, das der Server vorher dort registriert hat. Für dieses Objekt ruft der Client eine Methode auf. Gegebenenfalls kann das RMI-System auch einen Webserver nutzen, um Bytecode für Objekte vom Server zum Client bzw. umgekehrt zu übertragen.

Die Zusammenhänge, Begriffe und die erforderlichen Klassen und Methoden zur Entwicklung einer RMI-Anwendung werden nun im Folgenden anhand eines ersten Beispiels schrittweise erläutert.

Programm 7.1

Programm 7.1 zeigt die Implementierung eines *Echo-Dienstes*.

Remote Interface

Das *Remote Interface* definiert die Sicht des Client auf das entfernte Objekt. Dieses Interface enthält die Methoden, die für dieses Objekt entfernt aufgerufen werden können.

Ein *Remote Interface* ist von dem Interface

```
java.rmi.Remote
```

abgeleitet: `Remote` dient dazu, Interfaces zu kennzeichnen, deren Methoden entfernt aufgerufen werden sollen. `Remote` muss von allen Interfaces erweitert werden, die entfernte Methoden deklarieren.

RemoteException

`java.rmi.RemoteException` ist von `java.io.IOException` abgeleitet und ist Superklasse einer Reihe von Ausnahmen, die beim Aufruf einer entfernten Methode bei Netz- bzw. Protokollfehlern ausgelöst werden können. Jede Methode eines von `Remote` abgeleiteten Interfaces (*Remote Interface*) muss diese Klasse in der `throws`-Klausel aufführen.

Remote Interface Echo

```
import java.rmi.*;

public interface Echo extends Remote {
  String getEcho(String s) throws RemoteException;
}
```

Remote Object

Jedes Objekt, dessen Klasse ein Remote Interface implementiert, ist ein so genanntes entferntes Objekt (*Remote Object*), d.h. es implementiert die vorgeschriebenen entfernt aufrufbaren Methoden. Diese Methoden und die Konstruktoren müssen `RemoteException` in der `throws`-Klausel enthalten.

Exportieren

Damit eine Verbindung zwischen Client und Server aufgenommen werden kann und Methoden des Objekts entfernt aufgerufen werden können, muss das entfernte Objekt *exportiert* und damit *remote-fähig* gemacht werden.

Dies geschieht durch Ableiten von der Klasse

`java.rmi.server.UnicastRemoteObject`.

Bei Erzeugung des entfernten Objekts wird der Konstruktor dieser Superklasse aufgerufen, der das Objekt *exportiert*. Das exportierte Objekt kann nun über TCP/IP eingehende Nachrichten erhalten.

Implementierung des Remote Interface: EchoImpl

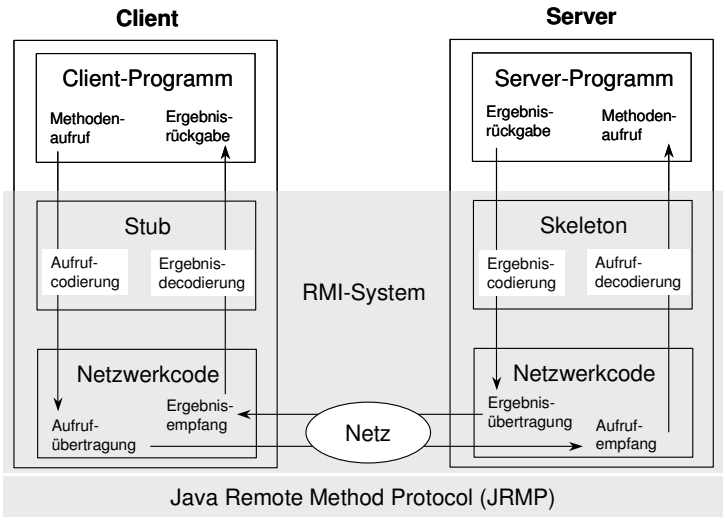
Es hat sich die Konvention durchgesetzt, dass die Implementierung des Interfaces `Xxx` in der Klasse `XxxImpl` erfolgt.

```
import java.rmi.*;
import java.rmi.server.*;

public class EchoImpl extends UnicastRemoteObject implements Echo {
  public EchoImpl() throws RemoteException { }

  public String getEcho(String s) throws RemoteException {
    return s;
  }
}
```

Stub und Skeleton

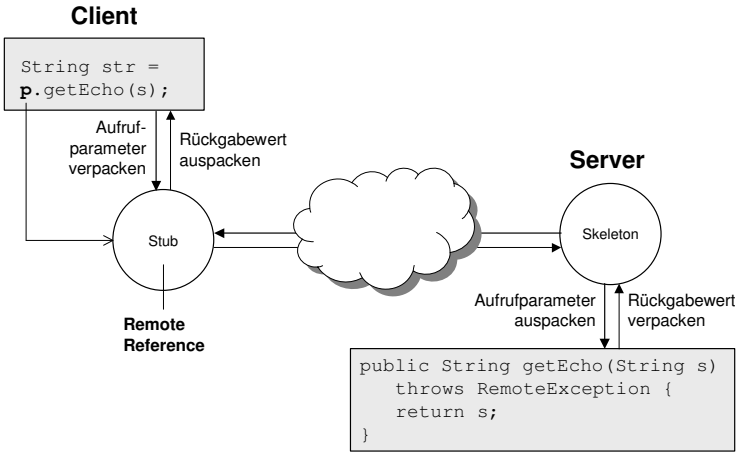
Die Codierung bzw. Decodierung der Aufrufparameter und Rückgabewerte von Methodenaufrufen und die Übermittlung der Daten zwischen Client und Server wird vom RMI-System und von generierten Klassen, beim Client *Stub* und beim Server *Skeleton* genannt, geregelt (siehe Bild 7.2).

***Bild 7.2**: Aufruf einer entfernten Methode*



Ein Stub-Objekt fungiert als *lokaler Stellvertreter* (Proxy) des entfernten Objekts. Ein Stub implementiert dasselbe Remote Interface, das auch die Klasse des entfernten Objekts implementiert.

***Bild 7.3**: Remote Reference*



Remote Reference

Ein Client erhält Zugriff auf ein entferntes Objekt durch eine so genannte *entfernte Referenz* (Remote Reference). Diese wird beim Erzeugen des lokalen Stub-Objekts bereitgestellt. Sie kapselt Informationen, die für den Zugriff auf das entfernte Objekt benötigt werden. Der Client ruft eine Methode des Stub-Objekts auf, die dann für den Methodenaufruf des entfernten Objekts auf der Serverseite sorgt (siehe Bild 7.3). Kurz gesagt steht also eine entfernte Referenz in Form eines Stub-Objekts zur Verfügung.

Generierung von Stub und Skeleton

Das *Java Remote Method Protocol* (JRMP) wird vom Stub genutzt, um mit dem Server zu kommunizieren. Es liegt in zwei Versionen vor: 1.1 und 1.2.

- Version 1.1 kommuniziert mit einem Skeleton (wird genutzt für Clients, die unter JDK 1.1 laufen),
- Version 1.2 benötigt keine Skeleton-Klasse.

Das Tool *rmic* erzeugt Stubs und Skeletons aus den kompilierten Klassen, die die Implementierung des Remote Interface enthalten.

- *rmic -v1.1 …* erzeugt Stubs und Skeletons für JRMP 1.1,
- *rmic -v1.2 …* erzeugt nur Stubs für JRMP 1.2.

Ab Java SE 5.0 werden Stubs zur Laufzeit dynamisch generiert, sodass *rmic* nicht mehr gebraucht wird. *rmic* steht aber weiterhin zur Verfügung, um Clients unter früheren Java-Versionen zu unterstützen. Der Aufruf von *rmic* ohne weitere Option verhält sich wie *rmic -v1.2 …* Um Stubs und Skeletons, die mit JRMP 1.1 und 1.2 kompatibel sein sollen, zu generieren, muss *rmic -vcompat …* genutzt werden.

Die Klasse `EchoServer` (siehe unten) erzeugt ein entferntes Objekt vom Typ `EchoImpl` und registriert dieses bei einem Namensdienst (Registry).

Registry

Das vom SDK bereitgestellte Programm *rmiregistry* stellt einen einfachen Dienst zur Verfügung, der es dem Client erlaubt, eine *erste Referenz* auf ein entferntes Objekt als *Einstiegspunkt* zu erhalten. Weitere Referenzen auf andere entfernte Objekte können von hier aus dann anwendungsspezifisch, z.B. als Rückgabewerte von entfernten Methodenaufrufen, geliefert werden.

Jeder Eintrag in der Registry besteht aus einem *Namen* und einer *Objektreferenz*.

Der Name hat die Form eines URL:

```
//host:port/Dienstname
```

Bis auf `Dienstname` können alle Bestandteile entfallen. Der Rechnername ist in diesem Fall `localhost` und die Portnummer 1099.

Soll für die Registry eine andere als die standardmäßig vorgesehene Portnummer 1099 benutzt werden, so muss sie als Aufrufparameter beim Start von *rmiregistry* angegeben werden.

Naming

Die Klasse `java.rmi.Naming` wird von Clients und Servern benutzt, um mit der Registry zu kommunizieren.

bind

```
static void bind(String name, Remote obj)
    throws java.rmi.AlreadyBoundException,
    java.net.MalformedURLException, java.rmi.RemoteException
```

registriert einen Eintrag für ein entferntes Objekt. `name` wird an `obj` gebunden. `AlreadyBoundException` wird ausgelöst, wenn `name` bereits eingetragen ist. `MalformedURLException` wird ausgelöst, wenn der Aufbau von `name` nicht korrekt ist.

rebind

```
static void rebind(String name, Remote obj)
    throws java.net.MalformedURLException, java.rmi.RemoteException
```

registriert einen Eintrag für ein entferntes Objekt. `name` wird an `obj` gebunden. Besteht bereits ein Eintrag zu diesem Namen, so wird der bestehende Eintrag überschrieben. `MalformedURLException` wird ausgelöst, wenn der Aufbau von `name` nicht korrekt ist.

unbind

```
static void unbind(String name)
    throws java.rmi.NotBoundException,
    java.net.MalformedURLException, java.rmi.RemoteException
```

entfernt den Eintrag zu `name`. `NotBoundException` wird ausgelöst, wenn zu `name` kein Eintrag vorhanden ist. `MalformedURLException` wird ausgelöst, wenn der Aufbau von `name` nicht korrekt ist.

Diese drei `Naming`-Methoden können nur auf dem Rechner ausgeführt werden, auf dem auch der Namensdienst läuft.

Der Server: EchoServer

```
import java.rmi.*;

public class EchoServer {
  public static void main(String args[]) throws Exception {
    Remote remote = new EchoImpl();
    Naming.rebind("echo", remote);
    System.out.println("EchoServer gestartet ...");
  }
}
```

Das RMI-System sorgt dafür, dass der Server läuft, auch wenn die Ausführung der `main`-Methode beendet ist.

Der Client: EchoClient

```
import java.rmi.*;

public class EchoClient {
  public static void main(String args[]) throws Exception {
    if (args.length != 2) {
      System.err.println("java EchoClient <host> <text>");
      System.exit(1);
    }

    String host = args[0];
    String text = args[1];

    Echo remote = (Echo) Naming.lookup("//" + host + "/echo");
    String received = remote.getEcho(text);
    System.out.println(received);
  }
}
```

lookup

Mittels der `Naming`-Methode `lookup` erhält der Client das Stub-Objekt zum entfernten Objekt, das unter dem Namen "`echo`" in der Registry eingetragen ist.

```
static Remote lookup(String name)
    throws java.rmi.NotBoundException,
    java.net.MalformedURLException, java.rmi.RemoteException
```

liefert die Referenz auf das Stub-Objekt für das unter `name` eingetragene entfernte Objekt. `MalformedURLException` wird ausgelöst, wenn der Aufbau von `name` nicht korrekt ist. `NotBoundException` wird ausgelöst, wenn zu `name` kein Eintrag vorhanden ist.

Test

Aufruf der Registry:

```
start /Dbuild rmiregistry
```

oder

```
start /Dbuild rmiregistry -J-Djava.rmi.server.logCalls=true
```

(Dies bewirkt, dass der Kommunikationsfluss zwischen Client und Server protokolliert wird.)

Aufruf des Servers:

```
start java -cp build EchoServer
```

Aufruf des Client:

```
java -cp build EchoClient localhost Hallo
```

Um sich gegen Einschleusen schädlichen Codes in den Client zu schützen, kann die Ausführung von einem *Security Manager* kontrolliert werden. Hierzu muss eine *Policy-Datei* mit beispielsweise folgendem Inhalt angelegt werden:

policy.txt

```
grant {
  permission java.net.SocketPermission "*:1024-", "connect";
};
```

Der Client ist dann wie folgt aufzurufen (in einer Zeile einzugeben):

```
java -Djava.security.manager Djava.security.policy=policy.txt
-cp build EchoClient localhost Hallo
```

Client und Server können natürlich auch auf unterschiedlichen Rechnern installiert und getestet werden.

LocateRegistry

rmiregistry startet die Registry auf einem Rechner, die dann für mehrere RMI-Server genutzt werden kann. Eine individuelle Registry kann aber auch innerhalb des Servers gestartet werden. Dazu steht die Klasse `LocateRegistry` im Paket `java.rmi.registry` zur Verfügung.

Die Methode

```
static Registry createRegistry(int port)
    throws java.rmi.RemoteException
```

erzeugt eine Registry die auf dem Port `port` Anfragen akzeptiert.

Aufrufparameter und Rückgabewert

Die Aufrufparameter und der Rückgabewert einer entfernten Methode können von einem einfachen Datentyp, Referenzen auf "normale" lokale Objekte oder Referenzen auf entfernte Objekte sein.

Übertragungsregeln

Für *entfernte Methoden* gelten die folgenden Übertragungsregeln:

Werte von einfachem Datentyp (z.B. `int`, `double`) werden wie bei lokalen Methodenaufrufen *by value* übertragen.

Lokale Objekte werden serialisiert, übertragen und vom Server deserialisiert. Dafür sorgen Stub und Skeleton. Lokale Objekte werden, anders als beim lokalen Methodenaufruf, als Kopie *by value* übertragen. Diese Objekte müssen also das Interface `java.io.Serializable` implementieren.

Exportierte *entfernte Objekte* werden *by reference* übertragen, d.h. es werden die Stub-Objekte, nicht Kopien der Originale übertragen.

Die folgende Tabelle fasst die Regeln zusammen:

Typ	lokale Methode	entfernte Methode
einfacher Typ	by value	by value
Objekt	by reference	by value (Serialisierung)
entferntes Objekt	by reference	by remote reference (Stub-Objekt)

7.2 Dienstauskunft

Das folgende Programm gibt eine Liste aller in der Registry gebundenen Namen aus.

Die `Naming`-Methode

```
static String[] list(String name)
    throws java.rmi.RemoteException, java.net.MalformedURLException
```

liefert ein Array von Namen, die an entfernte Objekte gebunden sind. `name` spezifiziert die Registry, also z.B. "`//localhost`".

Programm 7.2

```
import java.rmi.*;

public class ListRegistry {
  public static void main(String args[]) throws Exception {
    String registryName = args[0];

    String list[] = Naming.list(registryName);
    for (String name : list) {
      System.out.println(name);
    }
  }
}
```

Test

```
java -cp build ListRegistry //localhost
```

7.3 Transport by reference

Beim Aufruf entfernter Methoden werden lokale Objekte als Kopie in serialisierter Form übertragen. Das folgende Beispiel zeigt eine entfernte Methode, die als Rückgabewert eine *entfernte Referenz* liefert, mit deren Hilfe eine entfernte Methode eines weiteren entfernten Objekts vom Client aufgerufen werden kann.

Programm 7.3

Der RMI-Server soll Konten mit den Attributen Kontonummer (id), PIN und Saldo verwalten. Der Client kann ein neues Konto mit Angabe von Kontonummer und PIN anlegen oder ein bereits unter seiner Kontonummer eingerichtetes Konto öffnen. Für dieses Konto kann er dann einen Betrag einzahlen oder abheben sowie sich seinen Kontostand anzeigen lassen.

Für dieses Beispiel nutzen wir eine eigene `Exception`-Klasse:

KontoException

```
public class KontoException extends Exception {
  public KontoException() { }

  public KontoException(String msg) {
    super(msg);
  }
}
```

Remote Interface Konto

```
import java.rmi.*;

public interface Konto extends Remote {
  int getSaldo() throws RemoteException;
  void add(int betrag) throws RemoteException, KontoException;
}
```

Remote Interface KontoManager

```
import java.rmi.*;

public interface KontoManager extends Remote {
  Konto getKonto(int id, int pin) throws RemoteException,
    KontoException;
}
```

Die entfernte Methode `getKonto` gibt eine *entfernte Referenz* auf ein `Konto`-Objekt zurück.

KontoImpl

Wird versucht, das Konto zu überziehen, so wird eine Ausnahme vom Typ `KontoException` ausgelöst.

```
import java.rmi.*;
import java.rmi.server.*;

public class KontoImpl extends UnicastRemoteObject implements Konto {
  private int id;
  private int pin;
  private int saldo;

  public KontoImpl(int id, int pin) throws RemoteException {
    this.id = id;
    this.pin = pin;
  }
```

```
  public int getSaldo() throws RemoteException {
    return saldo;
  }

  public void add(int betrag) throws RemoteException, KontoException {
    if (saldo + betrag < 0) {
      throw new KontoException(
         "Das Konto kann nicht ueberzogen werden.");
    }
    saldo += betrag;
  }

  public int getPin() {
    return pin;
  }
}
```

KontoManagerImpl

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

public class KontoManagerImpl extends UnicastRemoteObject
  implements KontoManager {

  private Hashtable<Integer, KontoImpl> hashtable;

  public KontoManagerImpl() throws RemoteException {
    hashtable = new Hashtable<Integer, KontoImpl>();
  }

  public Konto getKonto(int id, int pin) throws RemoteException,
    KontoException {

    KontoImpl konto = hashtable.get(id);
    if (konto == null) {
      konto = new KontoImpl(id, pin);
      hashtable.put(id, konto);
      System.out.println("Konto " + id + " wurde eingerichtet.");
      return konto;
    }
    else {
      if (konto.getPin() == pin)
        return konto;
      else
        throw new KontoException("PIN ist ungueltig.");
    }
  }
}
```

Die Konten werden in einer *Hashtable* unter der jeweiligen Kontonummer gespeichert.

Ist bereits unter der angegebenen Kontonummer ein Konto vorhanden, so wird geprüft, ob die angegebene PIN gültig ist. Ist die PIN nicht korrekt, wird eine Ausnahme ausgelöst. Ist die

Kontonummer neu, so wird ein neues Konto eingerichtet. Bei fehlerfreier Verarbeitung wird in beiden Fällen die entfernte Referenz auf ein `Konto`-Objekt zurückgeliefert.

BankServer

```
import java.rmi.*;

public class BankServer {
  public static void main(String args[]) throws Exception {
    Remote remote = new KontoManagerImpl();
    Naming.rebind("bank", remote);
    System.out.println("BankServer gestartet ...");
  }
}
```

Beim Aufruf des Client werden als Parameter u.a. Kontonummer und PIN mitgegeben. Das Programm ermöglicht die Ausführung verschiedener Aktionen:

`get`	Anzeige des Saldos
`+nnn`	Betrag `nnn` einzahlen
`-nnn`	Betrag `nnn` auszahlen
`q`	Beenden

BankClient

```
import java.rmi.*;
import java.io.*;

public class BankClient {
  public static void main(String args[]) throws Exception {
    if (args.length != 3) {
      System.err.println("java BankClient <host> <id> <pin>");
      System.exit(1);
    }

    String host = args[0];
    int id = Integer.parseInt(args[1]);
    int pin = Integer.parseInt(args[2]);

    KontoManager manager = (KontoManager) Naming.lookup(
      "//" + host + "/bank");
    Konto konto = manager.getKonto(id, pin);

    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));

    String input;
    while (true) {
      try {
        System.out.println("Kommando eingeben (get | <zahl>| q):");
        input = in.readLine();
```

```
        if (input == null || input.length() == 0 || input.equals("q"))
          break;
        if (input.equals("get")) {
          System.out.println("Aktueller Kontostand: " +
            konto.getSaldo());
        }
        else {
          int betrag = Integer.parseInt(input);
          konto.add(betrag);
        }
      }
      catch (NumberFormatException e) { }
      catch (KontoException e) {
        System.out.println(e.getMessage());
      }
    }
  }
}
```

Mit `lookup` erhält der Client Zugriff auf das entfernte `KontoManager`-Objekt. Hierfür ruft er die Methode `getKonto` auf und erhält als Rückgabewert eine entfernte Referenz auf ein `Konto`-Objekt, für das er dann die entfernten `Konto`-Methoden aufruft.

Registry und Server werden wie in Kapitel 7.1 aufgerufen. *Test*

Aufruf des Client:

```
java -cp build BankClient localhost 1 1234
```

```
Kommando eingeben (get | <zahl>| q):
1000
Kommando eingeben (get | <zahl>| q):
get
Aktueller Kontostand: 1000
Kommando eingeben (get | <zahl>| q):
-2000
Das Konto kann nicht ueberzogen werden.
Kommando eingeben (get | <zahl>| q):
q
```

7.4 Mobile Agenten

Ein (serialisierbares) Objekt kann auch dann vom Client zum Server transportiert werden, wenn der Server den Bytecode der zugehörigen Klasse noch nicht zur Verfügung hat. Der Code muss dann ebenfalls zur Laufzeit ad hoc übertragen werden.

Wir nutzen hier diese Möglichkeit, um Methoden dieser Klasse auf dem Server *lokal* auszuführen. Derartige Nachrichten entsprechen von ihrem Wesen her einem *mobilen Agenten*, der im

Auftrag eines Client bestimmte Aufgaben auf einem anderen Rechner erledigt und danach zurückkehrt.

Programm 7.4

Wir entwickeln einen Server, der beliebige Aufgaben vom Client entgegennimmt, diese ausführt und die Ergebnisse zurückliefert. Das ist z. B. dann hilfreich, wenn der Server auf einer sehr schnellen Maschine läuft und komplexe mathematische Berechnungen ausgeführt werden müssen.

Eine Aufgabe kann durch ein beliebiges Objekt repräsentiert werden, dessen Klasse das Interface `Agent` implementiert:

Agent

```
public interface Agent extends java.io.Serializable {
  void execute();
}
```

ServerAgent

```
import java.rmi.*;

public interface ServerAgent extends Remote {
  Agent execute(Agent agent) throws RemoteException;
}
```

Die entfernte Methode `execute` des Remote Interface `ServerAgent` initiiert den Transport des `Agent`-Objekts und ruft die `Agent`-Methode `execute` auf:

ServerAgentImpl

```
import java.rmi.*;
import java.rmi.server.*;

public class ServerAgentImpl extends UnicastRemoteObject
  implements ServerAgent {

  public ServerAgentImpl() throws RemoteException { }

  public Agent execute(Agent agent) throws RemoteException {
    agent.execute();
    return agent;
  }
}
```

Server

```
import java.rmi.*;

public class Server {
  public static void main(String args[]) throws Exception {
    Remote remote = new ServerAgentImpl();
    Naming.rebind("agent", remote);
    System.out.println("Server gestartet ...");
  }
}
```

Die Klasse `DemoAgent` implementiert das Interface `Agent`. Die Methode `execute` berechnet die Summe der Zahlen von `1` bis zu einer vorgegebenen Zahl `n`. Diese Klasse soll später über das Netz zum Server transportiert werden.

DemoAgent

```
public class DemoAgent implements Agent {
  private int n;
  private int sum;

  public DemoAgent(int n) {
    this.n = n;
  }

  public void execute() {
    for (int i = 1; i <= n; i++) {
      sum += i;
    }
  }

  public int getResult() {
    return sum;
  }
}
```

Client

```
import java.rmi.*;

public class Client {
  public static void main(String args[]) throws Exception {
    String host = args[0];

    ServerAgent remote = (ServerAgent) Naming.lookup(
      "//" + host + "/agent");
    Agent demo = new DemoAgent(100);
    DemoAgent result = (DemoAgent) remote.execute(demo);
    System.out.println(result.getResult());
  }
}
```

Zum Test werden für Client und Server verschiedene Verzeichnisse eingerichtet. Die Klasse `DemoAgent` ist so für den Server lokal nicht erreichbar.

Um den Bytecode von `DemoAgent` herunterladen zu können, nutzt der Server einen HTTP-Server. Hier kann ein beliebiger Webserver genutzt werden. Ein einfacher HTTP-Server, der auf Anfrage Bytecode liefert, ist im Online-Service zu diesem Programmbeispiel vorhanden.

Für den Start des Client ist der URL für den zu übertragenden Bytecode als Wert der Property

```
java.rmi.server.codebase
```

anzugeben. Diese Information wird zum Client übertragen, sodass dieser dann die geeignete HTTP-Anfrage stellen kann.

Der Server muss einen *Security Manager* nutzen, da das Laden von Bytecode im Allgemeinen eine unsichere Aktivität ist.

Die *Policy-Datei* hat für unsere Zwecke den folgenden Inhalt:

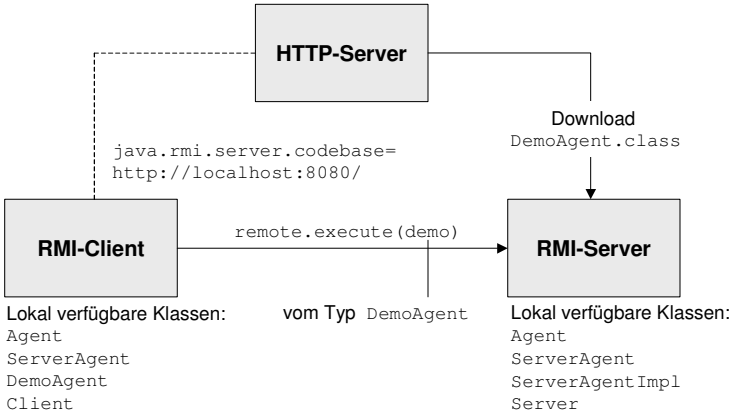
policy.txt

```
grant {
  permission java.net.SocketPermission "*:1024-", "connect,accept";
  permission java.net.SocketPermission "*:8080", "connect";
};
```

Unser HTTP-Server wird an die Portnummer 8080 gebunden werden, die somit bei der obigen Festlegung berücksichtigt ist.

Bild 7.4 zeigt die Konfiguration. Insbesondere geht aus der Abbildung hervor, welche Klassen dem Client, welche dem Server lokal zur Verfügung stehen.

***Bild 7.4**: Dynamisches Laden einer Klasse*



Test

Aufruf von Registry und Server (jeweils in einer Zeile einzugeben):

```
start /Dbuild rmiregistry
```

```
start java -Djava.security.manager -Djava.security.policy=policy.txt
-cp build Server
```

Aufruf des im Online-Service vorhandenen HTTP-Servers (im Verzeichnis `../client/build` befindet sich der Bytecode `DemoAgent.class`):

```
start java -cp build HTTPServer 8080 ../client/build
```

Aufruf des Client (in einer Zeile einzugeben):

```
java -Djava.rmi.server.codebase=http://localhost:8080/ -cp build
Client localhost
```

7.5 Callbacks

In diesem Kapitel sehen wir, dass ein Client auch zeitweise selbst Dienste anbieten kann. Der Server ruft eine entfernte Methode des Client auf.

Polling oder Callback

Eine typische Anwendungssituation ist: Der Client will Ereignisse beobachten, die auf dem Server eintreten. Statt nun regelmäßig in bestimmten Abständen eine Anfrage an den Server zu stellen, ob das interessierende Ereignis eingetreten ist oder nicht (*Polling*), lässt sich der Client vom Server über das Eintreten des Ereignisses informieren (*Callback*).

Damit dieser *Callback-Mechanismus* funktioniert, muss der Client sich beim Server registrieren (der Server speichert eine entfernte Referenz auf ein Remote-Objekt des Client). Dann kann der Server bei Eintreten des Ereignisses eine entfernte Methode des Client aufrufen. Bei dieser Lösung fällt im Vergleich zum Polling unnötige Rechenzeit und Netzlast weg.

Programm 7.5

Zur Veranschaulichung dieses Mechanismus entwickeln wir eine Anwendung (Client und Server), mit der Textnachrichten, die ein so genannter *Publisher* veröffentlicht, an interessierte Abonnenten (*Subscriber*) gesendet werden können. Der Server hat die Aufgabe, eine an ihn gerichtete Nachricht sofort an alle Abonnenten weiterzuleiten. Zu diesem Zweck muss er diese "kennen".

Textnachrichten werden in ein `Message`-Objekt verpackt, das zusätzlich den Zeitpunkt der Veröffentlichung enthält.

Message

```
public class Message implements java.io.Serializable {
  private long timestamp;
  private String text;

  public void setTimestamp(long timestamp) {
    this.timestamp = timestamp;
  }

  public long getTimestamp() {
    return timestamp;
  }
```

```
  public void setText(String text) {
    this.text = text;
  }

  public String getText() {
    return text;
  }
}
```

Das *entfernte Objekt des Servers* (vom Typ `MessageManager`) hat drei Methoden:

- `void setMessageListener(MessageListener listener)`
  meldet einen Subscriber an.
- `void removeMessageListener(MessageListener listener)`
  meldet einen Subscriber ab.
- `void send(Message msg)`
  sendet eine Nachricht.

Das *entfernte Objekt des Client* (vom Typ `MessageListener`) hat die Methode:

- `void onMessage(Message msg)`
  gibt die Nachricht aus.

MessageListener

```
import java.rmi.*;

public interface MessageListener extends Remote {
  void onMessage(Message msg) throws RemoteException;
}
```

MessageManager

```
import java.rmi.*;

public interface MessageManager extends Remote {
  void setMessageListener(MessageListener listener)
    throws RemoteException;
  void removeMessageListener(MessageListener listener)
    throws RemoteException;
  void send(Message msg) throws RemoteException;
}
```

Bild 7.5 zeigt den Zusammenhang.

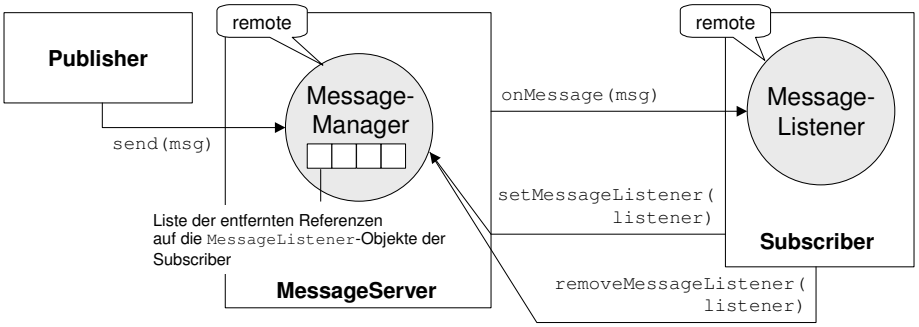


***Bild 7.5**: Callback-Mechanismus*

Wir implementieren zunächst den RMI-Server. Das `Vector`-Objekt `listeners` ist die Liste, die die entfernten Referenzen auf die `MessageListener`-Objekte der Abonnenten verwaltet. Die Methode `send` ruft für jede in der Liste gespeicherte Referenz die entfernte `MessageListener`-Methode `onMessage` auf. Wird hierbei (z.B. aufgrund des Abbruchs eines Subscribers) eine `RemoteException` ausgelöst, so wird die entsprechende Referenz aus der Liste entfernt.

Ein Thread gibt alle 5 Sekunden die Anzahl der Abonnenten am Bildschirm aus.

MessageManager-Impl

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

public class MessageManagerImpl extends UnicastRemoteObject
  implements MessageManager {

  private Vector<MessageListener> listeners;

  public MessageManagerImpl() throws RemoteException {
    listeners = new Vector<MessageListener>();
    new ControlThread().start();
  }

  public void setMessageListener(MessageListener listener)
    throws RemoteException {

    listeners.add(listener);
  }
```

```
  public void removeMessageListener(MessageListener listener)
    throws RemoteException {

    listeners.remove(listener);
  }

  public synchronized void send(Message msg) throws RemoteException {
    for (int i = listeners.size() - 1; i >= 0; i--) {
      MessageListener listener = listeners.get(i);
      try {
        listener.onMessage(msg);
      }
      catch (RemoteException e) {
        listeners.remove(listener);
      }
    }
  }

  private class ControlThread extends Thread {
    public void run() {
      while (true) {
        try {
          Thread.sleep(5000);
        }
        catch (InterruptedException e) { }

        System.out.println("Anzahl Subscribers: " + listeners.size());
      }
    }
  }
}
```

MessageServer

```
import java.rmi.*;

public class MessageServer {
  public static void main(String args[]) throws Exception {
    Remote remote = new MessageManagerImpl();
    Naming.rebind("message", remote);
    System.out.println("MessageServer gestartet ...");
  }
}
```

Nun implementieren wir *Publisher* und *Subscriber*.

Publisher

```
import java.rmi.*;

public class Publisher {
  public static void main(String args[]) throws Exception {
    if (args.length != 2) {
      System.err.println("java Publisher <host> <text>");
      System.exit(1);
    }
```

```
    String host = args[0];
    String text = args[1];

    MessageManager manager = (MessageManager) Naming.lookup(
      "//" + host + "/message");
    Message msg = new Message();
    msg.setTimestamp(System.currentTimeMillis());
    msg.setText(text);
    manager.send(msg);
  }
}
```

MessageListenerImpl

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import java.text.*;

public class MessageListenerImpl extends UnicastRemoteObject
  implements MessageListener {

  private SimpleDateFormat formatter;

  public MessageListenerImpl() throws RemoteException {
    formatter = new SimpleDateFormat("E, MMM d, yyyy HH:mm:ss z");
  }

  public void onMessage(Message msg) throws RemoteException {
    String timestamp = formatter.format(new Date(msg.getTimestamp()));
    System.out.println(timestamp);
    System.out.println(msg.getText());
  }
}
```

Subscriber

```
import java.rmi.*;

public class Subscriber {
  public static void main(String args[]) throws Exception {
    if (args.length != 2) {
      System.err.println("java Subscriber <host> <millis>");
      System.exit(1);
    }

    String host = args[0];
    int millis = Integer.parseInt(args[1]);

    MessageManager manager = (MessageManager) Naming.lookup(
      "//" + host + "/message");
    MessageListener listener = new MessageListenerImpl();
    manager.setMessageListener(listener);

    try {
      Thread.sleep(millis);
    }
```

```
    catch (InterruptedException e) { }

    manager.removeMessageListener(listener);
    System.exit(0);
  }
}
```

Test

Aufruf von Registry und Server:

```
start /Dbuild rmiregistry

start java -cp build MessageServer
```

Aufruf zweier Abonnenten:

```
start java -cp build Subscriber localhost 30000

start java -cp build Subscriber localhost 30000
```

Aufruf eines Publishers:

```
java -cp build Publisher localhost "Das ist ein Test"
```

7.6 Exkurs: RMI mit IIOP

CORBA

CORBA (*Common Object Request Architecture*), eine Spezifikation der *Object Management Group* (*OMG*), stellt eine Kommunikations- und Dienstinfrastruktur für verteilte objektorientierte Anwendungen bereit. Die Methodenaufrufe sind *sprachunabhängig*, d.h. Client und Server können mit unterschiedlichen Programmiersprachen implementiert werden. Schnittstellen werden mit der speziellen Beschreibungssprache *IDL* (*Interface Definition Language*) unabhängig von der Implementierungssprache definiert. Im Vergleich zu RMI ist CORBA jedoch komplizierter und aufwändiger in der Umsetzung.

IIOP

CORBA nutzt das Kommunikationsprotokoll *IIOP* (*Internet Inter-ORB Protocol*) auf der Basis von TCP/IP.

Neben dem Protokoll JRMP für eine reine Java-Umgebung (siehe Kapitel 7.1) unterstützt RMI auch IIOP (*Java RMI over IIOP*) und hat damit Zugang zu anderen CORBA-Anwendungen. *Enterprise JavaBeans* (*EJB*) der Plattform Java EE kommunizieren in der Regel über RMI/IIOP.

Im Folgenden wird gezeigt, wie eine einfache verteilte Anwendung auf der Basis von RMI mit IIOP implementiert werden kann. Analog zur RMI-Registry wird hier ein IIOP-fähiger Namensdienst, der vom *Object Request Broker Daemon* (orbd) angeboten wird, eingesetzt.

Programm 7.6

Interface LagerService

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface LagerService extends Remote {
  Lager getLager(String id) throws RemoteException;
}
```

Zu einer Artikelnummer *id* liefert die Methode `getLager` das zugehörige `Lager`-Objekt. Die Klasse `Lager` muss das Interface `java.io.Serializable` implementieren.

Die Klasse Lager

```
import java.util.Date;
import java.io.Serializable;

public class Lager implements Serializable {
  private String id;
  private int bestand;
  private Date datum;

  public Lager(String id, int bestand, Date datum) {
    this.id = id;
    this.bestand = bestand;
    this.datum = datum;
  }

  public String getId() {
    return id;
  }

  public void setId(String id) {
    this.id = id;
  }

  public int getBestand() {
    return bestand;
  }

  public void setBestand(int bestand) {
    this.bestand = bestand;
  }

  public Date getDatum() {
    return datum;
  }
```

```
  public void setDatum(Date datum) {
    this.datum = datum;
  }
}
```

Implementierung LagerServiceImpl

```
import java.rmi.RemoteException;
import javax.rmi.PortableRemoteObject;
import java.util.Hashtable;
import java.util.Date;

public class LagerServiceImpl extends PortableRemoteObject
  implements LagerService {

  private Hashtable<String, Lager> table;

  public LagerServiceImpl() throws RemoteException {
    table = new Hashtable<String, Lager>();
    Date datum = new Date();
    Lager[] list = new Lager[] {
      new Lager("4711", 100, datum),
      new Lager("4712", 80, datum),
      new Lager("4713", 55, datum)
    };
    for (int i = 0; i < list.length; i++) {
      table.put(list[i].getId(), list[i]);
    }
  }

  public Lager getLager(String id) throws RemoteException {
    return table.get(id);
  }
}
```

Da Client und Server über IIOP kommunizieren sollen, muss die Klasse von `javax.rmi.PortableRemoteObject` abgeleitet werden.

Der Server LagerServer

```
import javax.naming.InitialContext;
import javax.naming.Context;

public class LagerServer {
  public static void main(String args[]) throws Exception {
    LagerService service = new LagerServiceImpl();

    // Referenz im Naming Service mit JNDI veröffentlichen
    Context ctx = new InitialContext();
    ctx.rebind("LagerService", service);

    System.out.println("LagerServer gestartet ...");
  }
}
```

Das Programm erzeugt ein Service-Objekt und meldet dieses beim Namensdienst an. Auf den Namensdienst wird mit Hilfe des API *JNDI* (*Java Naming and Directory Interface*) zugegriffen. Hierzu werden das Interface `Context` und die Klasse `InitialContext`, beide aus dem Paket `javax.naming`, genutzt. der Service wird unter dem Namen "`LagerService`" eingetragen. Konkrete Angaben zum gewählten Namensdienst werden beim Aufruf des Programms über *Properties* eingestellt.

Der Client LagerClient

```
import javax.naming.InitialContext;
import javax.naming.Context;
import javax.rmi.PortableRemoteObject;

public class LagerClient {
  public static void main(String args[]) throws Exception {
    if (args.length != 1) {
      System.err.println("java LagerClient <id>");
      System.exit(1);
    }

    String id = args[0];

    // Referenz vom Naming Service mit JNDI erfragen
    Context ctx = new InitialContext();
    Object objref = ctx.lookup("LagerService");

    // Referenz casten
    LagerService service = (LagerService) PortableRemoteObject.narrow(
      objref, LagerService.class);

    Lager lager = service.getLager(id);

    if (lager == null) {
      System.out.println("Lager nicht vorhanden");
    }
    else {
      System.out.println(lager.getId());
      System.out.println(lager.getBestand());
      System.out.println(lager.getDatum());
    }
  }
}
```

Der Client kontaktiert den Namensdienst und erfragt mit `lookup` das als "`LagerService`" registrierte Service-Objekt. Die gelieferte Referenz muss mit `narrow` auf den entsprechenden Interface-Typ "gecastet" werden.

Compilierung

Nach Compilierung der Sourcen mit *javac* müssen mit *rmic* ein Stub und eine so genannte Tie-Klasse generiert werden:

```
rmic -classpath build -d build -iiop LagerServiceImpl
```

Mit Hilfe der zusätzlichen Option `-idl` können nach Bedarf auch die IDL-Beschreibungen zum Interface erzeugt werden.

Namensdienst starten

Der Namensdienst wird wie folgt auf Port 50000 gestartet:

```
start orbd -ORBInitialPort 50000
```

Start des Servers

```
start java -cp build LagerServer
```

Die im *build*-Verzeichnis abgelegte Datei *jndi.properties* enthält den Namen der JNDI-Treiberklasse für den Namensdienst sowie dessen URL:

```
java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
java.naming.provider.url=iiop://localhost:50000
```

Start des Client

```
java -cp build LagerClient 4711
```

Hier werden auch die Angaben in der Datei *jndi.properties* genutzt. Zur Ausführung werden die folgenden Bytecode-Dateien benötigt: `Lager.class`, `LagerService.class`, `LagerClient.class` und `_LagerService_Stub.class`.

Namensdienst, Server und Client können jeweils auf verschiedenen Rechnern im Netz laufen.

7.7 Aufgaben

1. Programmieren Sie einen RMI-Dienst, dessen entfernte Methode

   ```
   String getDaytime()
   ```

   die aktuelle Systemzeit des Servers liefert.

2. Aus einer Bücher-Datenbank sollen zu einer vorgegebenen Buchnummer Angaben zum Buch (Autor und Titel) über SQL abgefragt werden. Entwickeln Sie einen RMI-Dienst (inkl. Server) mit der Methode

   ```
   Buch getBuch(String id)
   ```

   und einen Client, der diese entfernte Methode aufruft.

   Die Klasse `Buch` soll die Angaben zum Buch als Attribute mit den entsprechenden set- und get-Methoden enthalten (vgl. auch Aufgabe 5 in Kapitel 4).

3. Erstellen Sie für den RMI-Server aus Kapitel 7.4 einen neuen *Agenten*, der zu einer vorgegebenen Zahl `n` die Fakultät `n!` ermittelt. Zur Berechnung kann die Klasse `java.math.BigInteger` genutzt werden.

4. Programmieren Sie ein Applet, das als *Subscriber* im Rahmen von Programm 7.5 Nachrichten in einer Textfläche (`JTextArea`) anzeigt. Hierzu muss `MessageListenerImpl` aus Programm 7.5 so angepasst werden, dass die Ausgabe in der Textfläche erscheint.

5. Entwickeln Sie nach der Vorlage in Kapitel 4.5 (Programm 4.3) ein auf RMI basierendes Chat-Programm: RMI-Server, RMI-Client (als eigenständige Applikation und als Applet). Der Client soll die gleiche Anwendungsfunktionalität und Benutzungsoberfläche haben wie das in Kapitel 4.5 entwickelte Programm. Nutzen Sie den Callback-Mechanismus.

6. Erstellen Sie einen universellen RMI-Server (`MultiServer`), der *mehrere Dienste gleichzeitig* anbieten kann. Zur Laufzeit sollen bestehende Dienste beendet und neue Dienste hinzugefügt werden können. Ebenso soll der Server kontrolliert beendet werden können.

   Der Server soll selbst den Dienst `MultiServerManager` mit den folgenden Methoden anbieten:

   ```
   void shutdown()
   void reconfigure()
   ```

   Die neu hinzuzufügenden Dienste sind in einer Textdatei in folgender Form (Beispiel) gespeichert:

   ```
   echo        EchoImpl
   daytime     DaytimeImpl
   ```

   Jede Zeile entspricht einem Dienst. Sie enthält den Dienstnamen und den Namen der Klasse, die den Dienst erbringt. Diese Datei ist bei der Rekonfiguration einzulesen.

   Für jeden Eintrag soll mittels Reflection (Nutzung der `Class`-Methoden `forName` und `newInstance`) ein neues entferntes Objekt erzeugt und bei der Registry angemeldet werden.

   Des Weiteren ist ein Client `MultiServerManagerClient` zu erstellen, der den Aufruf der Methoden `shutdown` und `reconfigure` ermöglicht.

8 Nachrichtendienste mit JMS

Die Kommunikation zwischen Client und Server in den vorangegangenen Kapiteln ist dadurch gekennzeichnet, dass die Teilnehmer *direkt* und *zeitgleich* miteinander in Verbindung treten. In der Regel ist der Client solange blockiert, bis der Server die Verarbeitung abgeschlossen und die Antwort zurückgesendet hat (*synchrone Kommunikation*).

MOM

Die Kommunikation auf der Basis von *nachrichtenorientierter Middleware* (*Message Oriented Middleware*, *MOM*) macht sich von dieser engen Kopplung frei. Der Austausch von Nachrichten erfolgt *asynchron* mit Hilfe eines Vermittlers (*Message Broker*, *MOM-Server*), der eine *Warteschlange* verwaltet. Die Nachricht des Senders wird vom Vermittler in die Warteschlange des Empfängers gelegt. Der Empfänger kann diese Nachricht zu einem späteren Zeitpunkt aus der Warteschlange holen. Sender und Empfänger agieren *unabhängig voneinander* und sind also über den Vermittler nur *lose gekoppelt*.

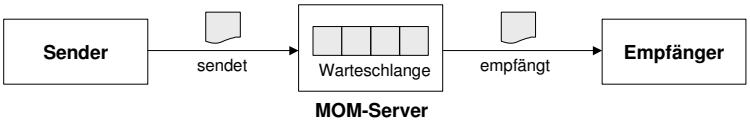


***Bild 8.1**: Nachrichtenaustausch*

Vorteile

Vorteile dieser asynchronen Kommunikation sind:

- Sender und Empfänger arbeiten unabhängig voneinander. Auch bei Ausfall eines Teils (Sender, Empfänger oder Vermittler) kann die Nachricht noch nachträglich zugestellt werden. Die beteiligten Komponenten können getrennt voneinander wieder gestartet werden. Daraus ergibt sich eine hohe *Fehlertoleranz*.
- Eine Nachricht kann dank des Vermittlers an *mehrere Empfänger* gesendet werden, ohne dass jeweils eine explizite Verbindung vom Client zu jedem Empfänger aufgebaut werden muss.
- Mehrere gesendete Nachrichten können gebündelt werden und zur *Steigerung der Effizienz* erst zu einem späteren Zeitpunkt in einem Rutsch abgeholt und verarbeitet werden.
- Durch Verwendung nachrichtenorientierter Middleware ist die Kommunikation *unabhängig von Programmiersprache und Plattform*.

- Die weitgehende Unabhängigkeit von Sender und Empfänger fördert den Einsatz dieser Technologie im Bereich der *Anwendungsintegration*.

8.1 Java Message Service

Java Message Service (*JMS*) wurde 1998 von Sun Microsystems veröffentlicht und ist inzwischen integraler Bestandteil der Java Enterprise Edition (Java EE). JMS ist ein API, das Syntax und Semantik für den Zugriff auf nachrichtenorientierte Middleware definiert. Verschiedene Hersteller bieten Implementierungen an, die dieser Spezifikation genügen. Jeder zu Java EE konforme Application Server muss über eine MOM-Implementierung verfügen.

Wir nutzen in diesem Kapitel

- JMS 1.1

  JAR-Datei und Dokumentation können über die Webseite

  ```
  http://java.sun.com/products/jms/
  ```

  heruntergeladen werden.

- JBoss Application Server 4.0.5

  JBoss ist frei verfügbar und kann über die Webseite

  ```
  http://www.jboss.org
  ```

  heruntergeladen werden.

- Alternativ kann auch die Open-Source-JMS-Implementierung *Apache ActiveMQ* eingesetzt werden. Sie kann über die Webseite

  ```
  http://www.activemq.org
  ```

  heruntergeladen werden. Näheres hierzu enthält der Online-Service.

Komponenten einer JMS-Anwendung

Eine JMS-Anwendung besteht aus einem JMS-Provider und mehreren JMS-Clients.

- *JMS-Provider*
  Der JMS-Provider ist der MOM-Server, in unserem Fall also der JBoss Application Server.
- JMS-*Clients*
  JMS-Clients sind die Java-Programme, die Nachrichten senden (*Message Producer*) bzw. empfangen (*Message Consumer*).

- *Messages*
  Messages (Nachrichten) haben ein festgelegtes Format und werden von JMS-Clients erzeugt, versandt und empfangen.
- *Administrierte Objekte*
  Administrierte Objekte werden vom JMS-Provider bereitgestellt und in einem Namensdienst veröffentlicht (siehe auch Kapitel 7.1). Diese Objekte werden von JMS-Clients über JNDI (Java Naming and Directory Interface) angefordert (siehe auch Kapitel 7.6), um Nachrichten senden oder empfangen zu können. Die *Verbindungsfabrik* (*ConnectionFactory*) und *Nachrichtenziele* (*Destinations*) sind administrierte Objekte.

Bild 8.2 zeigt die allgemeinen Schnittstellen des JMS-API. Je nach verwendetem Nachrichtenmodell (Point-to-Point, Publish-Subscribe) werden spezialisierte Schnittstellen eingesetzt. Die beiden Nachrichtenmodelle werden in späteren Abschnitten behandelt.

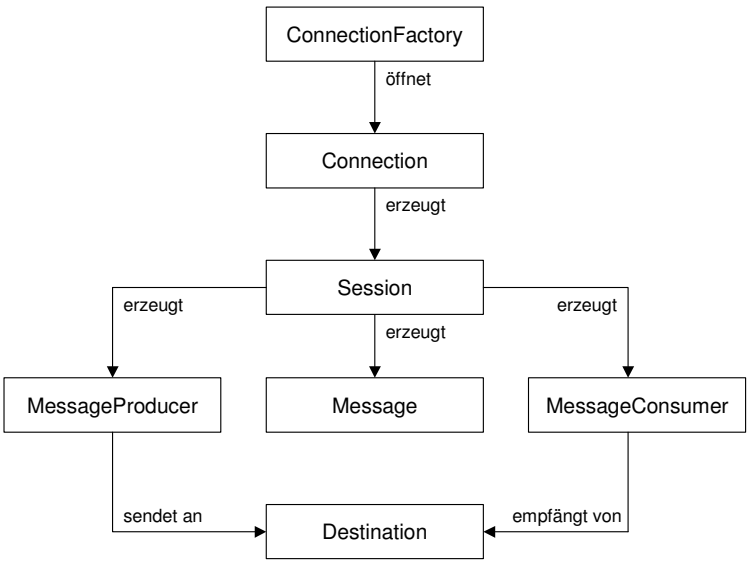


***Bild 8.2**: Wichtige JMS-Schnittstellen*

Die Interfaces und Klassen des JMS-API gehören alle zum Paket `javax.jms`.

Paket javax.jms

Im Folgenden werden die in Bild 8.2 aufgeführten Interfaces kurz beschrieben.

`ConnectionFactory`

Die Verbindungsfabrik dient dazu, Verbindungen zu einem JMS-Provider aufzubauen. Sie kann vom Administrator konfiguriert werden (administriertes Objekt) und wird in der Regel über einen Namensdienst zur Verfügung gestellt.

`Connection`

Eine Verbindung wird von einer Verbindungsfabrik erzeugt und stellt einen Kommunikationskanal zum JMS-Provider dar.

`Session`

Eine Sitzung wird von einer geöffneten Verbindung erstellt und repräsentiert einen Kontext, in dem Nachrichten, Sender oder Empfänger erzeugt werden.

`Message`

Eine Nachricht besteht aus einem Kopf (*Header*), Eigenschaften (*Properties*) und einem Rumpf (*Body*).

Der Kopf enthält verschiedene Felder, die zur Identifizierung und Verwaltung genutzt werden und zum Teil vom JMS-Provider automatisch gesetzt werden. Einige Header werden in den Beispielen dieses Kapitels verwendet.

Anwendungsbezogen können weitere Eigenschaften hinterlegt werden. Hierzu stehen die folgenden Methoden zur Verfügung:

```
void setXxxProperty(String name, xxx value) throws JMSException
xxx getXxxProperty(String name) throws JMSException
```

Hierbei steht `xxx` für `boolean`, `byte`, `short`, `int`, `long`, `float`, `double` oder `String`.

Der Rumpf speichert die Nutzdaten. Es existieren unterschiedliche Nachrichtentypen in Form von spezialisierten Interfaces:

`BytesMessage`, `MapMessage`, `ObjectMessage`, `StreamMessage` und `TextMessage`. Zwei dieser Typen werden in den Beispielen dieses Kapitels behandelt.

`MessageProducer`

Der *MessageProducer* hat die Aufgabe, Nachrichten an ein Ziel zu versenden.

`MessageConsumer`

Der *MessageConsumer* empfängt Nachrichten vom JMS-Provider

`Destination`

Ein Nachrichtenziel (`Destination`-Objekt) repräsentiert eine Warteschlange des JMS-Providers. Dieses kann vom Administrator eingerichtet (administriertes Objekt) und über einen Namensdienst bereitgestellt werden.

8.2 Das Point-to-Point-Modell

Beim *Point-to-Point-Modell* (*P2P*) wird eine vom Sender erzeugte Nachricht über eine *Queue* an *genau einen* Empfänger übermittelt (siehe Bild 8.1).

Sobald der Empfänger den Erhalt der Nachricht bestätigt hat, gilt sie als verbraucht und kann nicht erneut geholt werden. MOM-Server bieten das Konzept der persistenten Warteschlange. Nachrichten werden bis zu ihrer Auslieferung dauerhaft in einer Datenbank gespeichert, sodass sie ihren Empfänger auf jeden Fall erreichen, wenn er wieder aktiv ist.

In den Programmbeispielen dieses Abschnitts werden einfache Textnachrichten gesendet und empfangen.

Zunächst wird die Warteschlange für JBoss konfiguriert.

Queue bereitstellen

Die XML-Datei *myQueue1-service.xml* enthält die erforderlichen Angaben. Der Name der Queue ist *myQueue1*. Zugriff haben alle Benutzer mit der Rolle *guest*.

Inhalt von *myQueue1-service.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
  <mbean code="org.jboss.mq.server.jmx.Queue"
    name="jboss.mq.destination:service=Queue,name=myQueue1">
    <depends optional-attribute-name="DestinationManager">
      jboss.mq:service=DestinationManager
    </depends>
    <depends optional-attribute-name="SecurityManager">
      jboss.mq:service=SecurityManager
    </depends>
    <attribute name="SecurityConf">
      <security>
        <role name="guest" read="true" write="true"/>
      </security>
    </attribute>
  </mbean>
</server>
```

Diese Datei muss in das Verzeichnis

```
<JBoss-Home>\server\default\deploy\jms
```

kopiert werden bevor oder nachdem der Server mit

```
<JBoss-Home>\bin\run.bat
```

gestartet wurde.

Meldung des JBoss Application Servers:

```
INFO  [myQueue1] Bound to JNDI name: queue/myQueue1
```

Queue überwachen

Die Queue kann mittels der JMX-Konsole überwacht werden. Hierzu ist die folgende Seite im Browser aufzurufen:

```
http://localhost:8080/jmx-console/
```

Unter *jboss.mq.destination* findet man den entsprechenden Eintrag

```
name=myQueue1,service=Queue
```

Diverse Attribute und Operationen stehen zur Verfügung.

Zunächst werden die in den Programmbeispielen benutzten Typen und Methoden vorgestellt.

JMSException

Die meisten Methoden des JMS-API lösen bei Fehlern eine Ausnahme vom Typ `JMSException` (direkte Subklasse von `Exception`) aus.

Queue

`Queue` ist Subinterface von `Destination` und repräsentiert eine Warteschlange für das P2P-Modell.

QueueConnection-Factory

Das Interface `QueueConnectionFactory` ist ein Subinterface von `ConnectionFactory`.

```
QueueConnection createQueueConnection(String user, String password)
    throws JMSException
```

erzeugt eine Verbindung zu einer Queue.

Connection

```
void start() throws JMSException
```

startet die Auslieferung eingegangener Nachrichten.

```
void close() throws JMSException
```

schließt die Verbindung.

QueueConnection

`QueueConnection` ist Subinterface von `Connection` und repräsentiert eine Verbindung zu einer Queue im P2P-Modell.

`QueueSession` **`createQueueSession`**`(`
`boolean transacted, int acknowledgeMode) throws JMSException`
erzeugt eine Session. `transacted` gibt an, ob das Senden bzw. Empfangen von Nachrichten innerhalb einer Transaktion stattfinden soll. Im Beispiel wird keine Transaktionskontrolle genutzt. `acknowledgeMode` legt fest, wer für die Empfangsbestätigung zuständig ist. Im Beispiel wird `Session.AUTO_ACKNOWLEDGE` (automatische Bestätigung durch die Session) genutzt.

Session

`TextMessage` **`createTextMessage`**`() throws JMSException`
erzeugt eine Textnachricht.

`void` **`close`**`() throws JMSException`
schließt die Sitzung.

QueueSession

`QueueSession` ist Subinterface von `Session`.

`QueueSender` **`createSender`**`(Queue queue) throws JMSException`
erzeugt einen Sender für `queue`.

`QueueReceiver` **`createReceiver`**`(Queue queue) throws JMSException`
erzeugt einen Empfänger für `queue`.

MessageProducer

`void` **`setTimeToLive`**`(long timeToLive) throws JMSException`
setzt die Gültigkeitsdauer für Nachrichten in Millisekunden (`0` steht für unbegrenzt).

`void` **`close`**`() throws JMSException`
beendet den Message Producer.

QueueSender

`QueueSender` ist Subinterface von `MessageProducer`.

`void` **`send`**`(Message message) throws JMSException`
sendet die Nachricht `message`.

QueueReceiver

`QueueReceiver` ist Subinterface von `MessageConsumer`.

MessageConsumer

`Message` **`receive`**`() throws JMSException`
wartet auf das Eintreffen einer Nachricht und kehrt dann zurück.

`Message` **`receive`**`(long timeout) throws JMSException`
wartet maximal `timeout` Millisekunden auf das Eintreffen einer Nachricht und kehrt dann zurück.

`void` **`setMessageListener`**`(MessageListener listener) throws JMSException`
registriert ein Objekt vom Typ `MessageListener`, um asynchron auf eine Nachricht zu warten.

`void` **`close`**`() throws JMSException`
beendet den Message Consumer.

MessageListener

Dieses Interface definiert die Methode

`void` **`onMessage`**`(Message message)`

TextMessage

`TextMessage` ist Subinterface von `Message`.

`void` **`setText`**`(String string) throws JMSException`
setzt den Inhalt der Textnachricht.

`String` **`getText`**`() throws JMSException`
liefert den Inhalt der Textnachricht.

Programm 8.1

Es folgen die Programmbeispiele `Producer`, `Consumer1` und `Consumer2`.

Producer

```
import javax.jms.*;
import javax.naming.*;

public class Producer {
  private static final String DESTINATION = "queue/myQueue1";
  private static final String USER = "guest";
  private static final String PASSWORD = "guest";

  private QueueConnection connection;
  private QueueSession session;
  private QueueSender sender;
  private String text;
  private long expiration;

  public Producer(String text, long expiration)
    throws NamingException, JMSException {

    this.text = text;
    this.expiration = expiration;

    // JNDI-Kontext erzeugen
    Context ctx = new InitialContext();

    // ConnectionFactory über Namensdienst auslesen
    QueueConnectionFactory factory =
      (QueueConnectionFactory) ctx.lookup("ConnectionFactory");

    // Zieladresse über Namensdienst auslesen
    Queue queue = (Queue) ctx.lookup(DESTINATION);

    // Verbindung aufbauen
    connection = factory.createQueueConnection(USER, PASSWORD);

    // Session erzeugen
    session = connection.createQueueSession(
      false, Session.AUTO_ACKNOWLEDGE);

    // Sender erzeugen
    sender = session.createSender(queue);
  }
```

```
  // Nachricht erzeugen und senden
  public void sendMessage() throws JMSException {
    TextMessage message = session.createTextMessage();
    message.setText(text);
    sender.setTimeToLive(expiration);
    sender.send(message);
  }

  // Ressourcen freigeben
  public void close() throws JMSException {
    sender.close();
    session.close();
    connection.close();
  }

  public static void main(String[] args) throws Exception {
    String text = args[0];
    long expiration =
      (args.length == 2) ? Long.parseLong(args[1]) : 0;
    Producer producer = new Producer(text, expiration);
    producer.sendMessage();
    producer.close();
  }
}
```

Im folgenden Programm wartet der Empfänger aktiv auf das Eintreffen von Nachrichten (*Pull-Prinzip*). Nach *timeout* Millisekunden Wartezeit wird das Programm beendet.

Consumer1

```
import javax.jms.*;
import javax.naming.*;

public class Consumer1 {
  private static final String DESTINATION = "queue/myQueue1";
  private static final String USER = "guest";
  private static final String PASSWORD = "guest";

  private QueueConnection connection;
  private QueueSession session;
  private QueueReceiver receiver;
  private long timeout;

  public Consumer1(long timeout) throws NamingException,
    JMSException {

    this.timeout = timeout;

    // JNDI-Kontext erzeugen
    Context ctx = new InitialContext();

    // ConnectionFactory über Namensdienst auslesen
    QueueConnectionFactory factory =
      (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
```

```
    // Zieladresse über Namensdienst auslesen
    Queue queue = (Queue) ctx.lookup(DESTINATION);

    // Verbindung aufbauen
    connection = factory.createQueueConnection(USER, PASSWORD);

    // Session erzeugen
    session = connection.createQueueSession(
      false, Session.AUTO_ACKNOWLEDGE);

    // Empfänger erzeugen
    receiver = session.createReceiver(queue);

    // Empfang von Nachrichten starten
    connection.start();
  }

  // Aktives Warten auf Nachrichten (Pull-Prinzip)
  public void receiveMessage() throws JMSException {
    Message message;
    while ((message = receiver.receive(timeout)) != null) {
      if (message instanceof TextMessage) {
        TextMessage textMessage = (TextMessage) message;
        System.out.println(textMessage.getText());
      }
    }
  }

  // Ressourcen freigeben
  public void close() throws JMSException {
    receiver.close();
    session.close();
    connection.close();
  }

  public static void main(String[] args) throws Exception {
    long timeout = Long.parseLong(args[0]);
    Consumer1 consumer = new Consumer1(timeout);
    consumer.receiveMessage();
    consumer.close();
  }
}
```

Im Programm *Consumer2* wird der Empfänger vom MOM-Server durch die Callback-Methode `onMessage` über das Eintreffen einer Nachricht informiert (*Push-Prinzip*). Das Programm wird nach *timeout* Millisekunden beendet.

Consumer2

```
import javax.jms.*;
import javax.naming.*;

public class Consumer2 implements MessageListener {
  private static final String DESTINATION = "queue/myQueue1";
  private static final String USER = "guest";
  private static final String PASSWORD = "guest";
  private QueueConnection connection;
  private QueueSession session;
  private QueueReceiver receiver;

  public Consumer2() throws NamingException, JMSException {
    // JNDI-Kontext erzeugen
    Context ctx = new InitialContext();

    // ConnectionFactory über Namensdienst auslesen
    QueueConnectionFactory factory =
      (QueueConnectionFactory) ctx.lookup("ConnectionFactory");

    // Zieladresse über Namensdienst auslesen
    Queue queue = (Queue) ctx.lookup(DESTINATION);

    // Verbindung aufbauen
    connection = factory.createQueueConnection(USER, PASSWORD);

    // Session erzeugen
    session = connection.createQueueSession(
      false, Session.AUTO_ACKNOWLEDGE);

    // Empfänger erzeugen
    receiver = session.createReceiver(queue);

    // MessageListener setzen
    receiver.setMessageListener(this);

    // Empfang von Nachrichten starten
    connection.start();
  }

  // Nachrichten werden im Push-Verfahren empfangen
  public void onMessage(Message message) {
    try {
      if (message instanceof TextMessage) {
        TextMessage textMessage = (TextMessage) message;
        System.out.println(textMessage.getText());
      }
    }
    catch (JMSException e) {
      System.err.println(e);
    }
  }
```

```
  // Ressourcen freigeben
  public void close() throws JMSException {
    receiver.close();
    session.close();
    connection.close();
  }

  public static void main(String[] args) throws Exception {
    long timeout = Long.parseLong(args[0]);
    Consumer2 consumer = new Consumer2();
    Thread.sleep(timeout);
    consumer.close();
  }
}
```

Die Verbindungsdaten für den Zugriff auf den Namensdienst über JNDI sind in der Datei *jndi.properties* ausgelagert.

jndi.properties

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jnp.interfaces
```

Zur Compilierung wird die JAR-Datei *jms.jar*, zur Ausführung der Programme die JAR-Datei *jbossall-client.jar* aus der JBoss-Installation verwendet. *jndi.properties* muss in das Verzeichnis *build* kopiert werden.

Die Umgebungsvariablen `JMS_PATH` und `JBOSSCLIENT_PATH` müssen gesetzt werden:

```
set JMS_PATH=Verzeichnis\jms.jar
set JBOSSCLIENT_PATH=Verzeichnis\client\jbossall-client.jar
```

Test

```
javac -sourcepath src -cp %JMS_PATH% -d build src/*.java
```

```
java -cp build;%JBOSSCLIENT_PATH% Producer Hallo
```

```
java -cp build;%JBOSSCLIENT_PATH% Consumer1 5000
```

oder

```
java -cp build;%JBOSSCLIENT_PATH% Consumer2 30000
```

Programm 8.2 ermöglicht es, den Inhalt einer Warteschlange abzufragen, ohne die Nachrichten zu löschen. ***Programm 8.2***

Hierzu gibt es das Interface `QueueBrowser` mit den folgenden Methoden: *QueueBrowser*

`java.util.Enumeration` **`getEnumeration`**`() throws JMSException`
liefert ein `Enumeration`-Objekt zum sequentiellen Durchlaufen der in der Warteschlange enthaltenen Nachrichten.

`void` **`close`**`() throws JMSException`
schließt den Browser.

Mit der `QueueSession`-Methode

`QueueBrowser` **`createBrowser`**`(Queue queue) throws JMSException`

wird ein Browser für die Warteschlange `queue` erzeugt.

Header einer Nachricht können mit *get-* und *set*-Methoden des Interfaces `Message` abgefragt bzw. gesetzt werden. Programm 8.2 verwendet die folgenden automatisch gesetzten Header: *Nachrichtenkopf (Header)*

`JMSMessageID`

Eindeutiger Bezeichner zur Identifikation einer Nachricht (Typ `String`).

`JMSPriority`

Wert zwischen `0` (niedrig) und `9` (hoch), der die Priorität der Auslieferung einer Nachricht festlegt (Typ `int`).

`JMSTimestamp`

Zeitpunkt der Übergabe der Nachricht an den JMS-Provider in Millisekunden (Typ `long`).

`JMSExpiration`

Zeitpunkt des Verfalls einer Nachricht in Millisekunden (Typ `long`). Dieser kann mit der `MessageProducer`-Methode `setTimeToLive` (siehe oben) bestimmt werden.

QueueInfo

```
import javax.jms.*;
import javax.naming.*;
import java.util.Enumeration;
import java.util.Date;

public class QueueInfo {
  private static final String DESTINATION = "queue/myQueue1";
  private static final String USER = "guest";
  private static final String PASSWORD = "guest";
  private QueueConnection connection;
  private QueueSession session;
  private QueueBrowser browser;

  public QueueInfo() throws NamingException, JMSException {
    Context ctx = new InitialContext();
    QueueConnectionFactory factory =
      (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
    Queue queue = (Queue) ctx.lookup(DESTINATION);
    connection = factory.createQueueConnection(USER, PASSWORD);
    session = connection.createQueueSession(
      false, Session.AUTO_ACKNOWLEDGE);

    // QueueBrowser erzeugen
    browser = session.createBrowser(queue);
    connection.start();
  }

  public void list() throws JMSException {
    Enumeration e = browser.getEnumeration();
    Message message;
    int cnt = 0;
    while (e.hasMoreElements()) {
      cnt++;
      message = (Message) e.nextElement();

      System.out.print(cnt + ".");
      System.out.println("\tMessageID: " + message.getJMSMessageID());
      System.out.println("\tTimestamp: " +
        new Date(message.getJMSTimestamp()));
      System.out.println("\tPriority: " + message.getJMSPriority());
      long expiration = message.getJMSExpiration();
      if (expiration == 0)
        System.out.println("\tExpiration: 0");
      else
        System.out.println("\tExpiration: " + new Date(expiration));
      System.out.println();
    }
  }

  public void close() throws JMSException {
    browser.close();
    session.close();
    connection.close();
  }
```

```
  public static void main(String[] args) throws Exception {
    QueueInfo info = new QueueInfo();
    info.list();
    info.close();
  }
}
```

8.3 Das Request-Reply-Modell

Ein Spezialfall des *Point-to-Point-Modells* ist das *Request-Reply-Modell*, das eine synchrone Kommunikation zwischen Sender und Empfänger simuliert. Der Sender wartet solange, bis die Antwortnachricht eintrifft. Hierzu wird *eine temporäre Warteschlange* für die Antwortnachricht genutzt. Diese wird vom Sender für die Dauer der Kommunikation erzeugt und dem Empfänger über den Nachrichten-Header `JMSReplyTo` (Typ `Destination`) mitgeteilt.

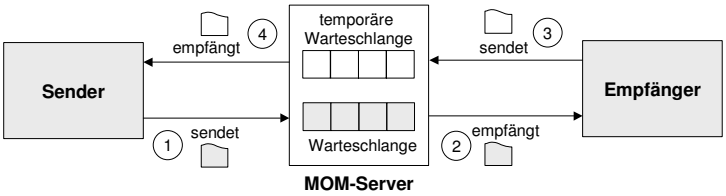


***Bild 8.3**: Request-Reply-Modell*

Das Interface `TemoraryQueue` ist Subinterface von `Queue` und enthält die Lösch-Methode

```
void delete() throws JMSException
```

Die folgende `QueueSession`-Methode erzeugt eine temporäre Warteschlange:

```
TemporaryQueue createTemporaryQueue() throws JMSException
```

Programm 8.3 bietet einen Echo-Dienst, der bereits aus vorhergehenden Kapiteln bekannt ist. ***Programm 8.3***

EchoClient

```
import javax.jms.*;
import javax.naming.*;

public class EchoClient {
  private static final String DESTINATION = "queue/myQueue1";
  private static final String USER = "guest";
  private static final String PASSWORD = "guest";

  private String text;
  private QueueConnectionFactory factory;
  private Queue queue;

  public EchoClient(String text) throws NamingException,
    JMSException {

    this.text = text;
    Context ctx = new InitialContext();
    factory = (QueueConnectionFactory) ctx.lookup(
      "ConnectionFactory");
    queue = (Queue) ctx.lookup(DESTINATION);
  }

  public void process() throws JMSException {
    QueueConnection connection = null;
    QueueSession session = null;
    TemporaryQueue tempQueue = null;
    QueueSender sender = null;
    QueueReceiver receiver = null;

    try {
      connection = factory.createQueueConnection(USER, PASSWORD);
      session = connection.createQueueSession(
        false, Session.AUTO_ACKNOWLEDGE);

      // temporäre Queue für die Antwort erzeugen
      tempQueue = session.createTemporaryQueue();

      sender = session.createSender(queue);
      receiver = session.createReceiver(tempQueue);
      connection.start();

      TextMessage request = session.createTextMessage();
      request.setText(text);
      request.setJMSReplyTo(tempQueue);
      sender.send(request);

      // auf Antwort warten
      TextMessage response = (TextMessage) receiver.receive();

      System.out.println(response.getText());
    }
    finally {
      sender.close();
      receiver.close();
      tempQueue.delete();
```

```
      session.close();
      connection.close();
    }
  }

  public static void main(String[] args) throws Exception {
    String text = args[0];
    EchoClient client = new EchoClient(text);
    client.process();
  }
}
```

EchoServer

```
import javax.jms.*;
import javax.naming.*;

public class EchoServer {
  private static final String DESTINATION = "queue/myQueue1";
  private static final String USER = "guest";
  private static final String PASSWORD = "guest";

  private QueueConnectionFactory factory;
  private Queue queue;

  public EchoServer() throws NamingException, JMSException {
    Context ctx = new InitialContext();
    factory = (QueueConnectionFactory) ctx.lookup(
      "ConnectionFactory");
    queue = (Queue) ctx.lookup(DESTINATION);
  }

  public void process() throws JMSException {
    QueueConnection connection = factory.createQueueConnection(
      USER, PASSWORD);
    QueueSession session = connection.createQueueSession(
      false, Session.AUTO_ACKNOWLEDGE);
    QueueReceiver receiver = session.createReceiver(queue);
    connection.start();
    System.out.println("EchoServer gestartet ...");

    while (true) {
      TextMessage request = (TextMessage) receiver.receive();
      String text = request.getText();
      Queue tempQueue = (Queue) request.getJMSReplyTo();
      TextMessage response = session.createTextMessage();
      response.setText(text);
      QueueSender sender = session.createSender(tempQueue);
      sender.send(response);
    }
  }

  public static void main(String[] args) throws Exception {
    EchoServer server = new EchoServer();
    server.process();
  }
}
```

Natürlich kann zur Verbesserung der Performance die clientspezifische Verarbeitung einer empfangenen Nachricht in einem eigenen Thread erfolgen (siehe Aufgabe 2).

8.4 Das Publish-Subscribe-Modell

Beim *Publish-Subscribe-Modell* (*Pub/Sub*) sendet der *Publisher* Nachrichten zu einem bestimmten Thema (*Topic*), die vom Vermittler (MOM-Server) in einer entsprechenden Warteschlange eingestellt werden. *Subscriber* können beim Vermittler ein bestimmtes Thema abonnieren. Alle Subscriber erhalten dann die Nachrichten, die zu diesem Thema veröffentlicht wurden. Im Unterschied zum P2P-Modell erhalten sie aber nur die Nachrichten, die während ihrer aktiven Verbindung mit dem Vermittler versandt wurden. Publisher und Subscriber sind vollständig unabhängig voneinander.

Es können jedoch auch dauerhafte Abonnements eingerichtet werden. Ein *dauerhafter Subscriber* erhält dann später auch die Nachrichten zu einem Thema, die veröffentlicht wurden, während er keine Verbindung zum Vermittler hatte (siehe Kapitel 8.5).

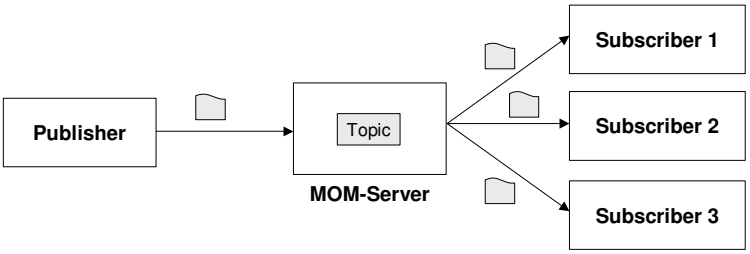


***Bild 8.4**: Pub/Sub-Modell*

Analog zum Nachrichtenziel *Queue* in Kapitel 8.2 existieren die folgenden Interfaces:

`Topic` (Subinterface von `Destination`)

`TopicConnectionFactory` (Subinterface von `ConnectionFactory`)

`TopicConnection` (Subinterface von `Connection`)

`TopicSession` (Subinterface von `Session`)

`TopicPublisher` (Subinterface von `MessageProducer`)

`TopicSubscriber` (Subinterface von `MessageConsumer`)

`TopicConnectionFactory`-Methode:

```
TopicConnection createTopicConnection(String user, String password)
    throws JMSException
```

`TopicConnection`-Methode:

```
TopicSession createTopicSession(
    boolean transacted, int acknowledgeMode) throws JMSException
```

`TopicSession`-Methoden:

```
TopicPublisher createPublisher(Topic topic) throws JMSException
TopicSubscriber createSubscriber(Topic topic) throws JMSException
```

`TopicPublisher`-Methode:

```
void publish(Message message) throws JMSException
```

MapMessage

Das folgende Programmbeispiel nutzt den Nachrichtentyp `MapMessage`. Eine solche Nachricht besteht im Nachrichtenrumpf aus Schlüssel-Wert-Paaren. Der Schlüssel ist vom Typ `String`, der Wert vom Typ `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`, `String` oder `byte[]`.

Zu jedem Datentyp existieren die *get*- und *set*-Methoden:

```
void setXxx(xxx value) throws JMSException
xxx getXxx(String name) throws JMSException
```

Für den Typ `byte[]` heißen die Methoden `setBytes` und `getBytes`.

Eine `MapMessage` wird mit der folgenden `Session`-Methode erzeugt:

```
MapMessage createMapMessage() throws JMSException
```

Programm 8.4

Der Publisher in Programm 8.4 veröffentlicht in regelmäßigen Abständen Messdaten mit Zeit- und Wertangabe. Subscriber werten diese Messdaten aus.

Zunächst muss das Thema (Topic) für JBoss konfiguriert werden.

Topic bereitstellen

Die XML-Datei *myTopic1-service.xml* enthält die erforderlichen Angaben. Der Topic-Name ist *myTopic1*. Zugriff haben alle Benutzer mit der Rolle *guest*.

Inhalt von *myTopic1-service.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
  <mbean code="org.jboss.mq.server.jmx.Topic"
    name="jboss.mq.destination:service=Topic,name=myTopic1">
    <depends optional-attribute-name="DestinationManager">
      jboss.mq:service=DestinationManager
    </depends>
    <depends optional-attribute-name="SecurityManager">
      jboss.mq:service=SecurityManager
    </depends>
    <attribute name="SecurityConf">
      <security>
        <role name="guest" read="true" write="true"/>
      </security>
    </attribute>
  </mbean>
</server>
```

Publisher

```
import javax.jms.*;
import javax.naming.*;
import java.util.*;
import java.text.*;

public class Publisher {
  private static final String DESTINATION = "topic/myTopic1";
  private static final String USER = "guest";
  private static final String PASSWORD = "guest";

  private TopicConnectionFactory factory;
  private Topic topic;

  public Publisher() throws NamingException, JMSException {
    Context ctx = new InitialContext();
    factory = (TopicConnectionFactory) ctx.lookup(
      "ConnectionFactory");
    topic = (Topic) ctx.lookup(DESTINATION);
  }

  public void process() throws JMSException {
    TopicConnection connection = factory.createTopicConnection(
      USER, PASSWORD);
    TopicSession session = connection.createTopicSession(
      false, Session.AUTO_ACKNOWLEDGE);
    TopicPublisher publisher = session.createPublisher(topic);

    SimpleDateFormat formatter = new SimpleDateFormat("HH:mm:ss");
    while (true) {
      String time = formatter.format(new Date());
      double value = Math.random() * 100;
      MapMessage message = session.createMapMessage();
      message.setString("Time", time);
      message.setDouble("Value", value);
      publisher.publish(message);
```

```
      try {
        Thread.sleep(2000);
      }
      catch (InterruptedException e) { }
    }
  }

  public static void main(String[] args) throws Exception {
    Publisher pub = new Publisher();
    pub.process();
  }
}
```

Subscriber

```
import javax.jms.*;
import javax.naming.*;

public class Subscriber implements MessageListener {
  private static final String DESTINATION = "topic/myTopic1";
  private static final String USER = "guest";
  private static final String PASSWORD = "guest";

  private TopicConnectionFactory factory;
  private Topic topic;
  private TopicConnection connection;
  private TopicSession session;
  private TopicSubscriber subscriber;

  public Subscriber() throws NamingException, JMSException {
    Context ctx = new InitialContext();
    factory = (TopicConnectionFactory) ctx.lookup(
      "ConnectionFactory");
    topic = (Topic) ctx.lookup(DESTINATION);
  }

  public void subscribe() throws JMSException {
    connection = factory.createTopicConnection(USER, PASSWORD);
    session = connection.createTopicSession(
      false, Session.AUTO_ACKNOWLEDGE);
    subscriber = session.createSubscriber(topic);
    subscriber.setMessageListener(this);
    connection.start();
  }

  public void close() throws JMSException {
    subscriber.close();
    session.close();
    connection.close();
  }
```

```
  public void onMessage(Message message) {
    try {
      if (message instanceof MapMessage) {
        MapMessage mapMessage = (MapMessage) message;
        System.out.println(mapMessage.getString("Time"));
        System.out.println(mapMessage.getDouble("Value"));
        System.out.println();
      }
    }
    catch (JMSException e) {
      System.err.println(e);
    }
  }

  public static void main(String[] args) throws Exception {
    long time = Long.parseLong(args[0]);
    Subscriber sub = new Subscriber();
    sub.subscribe();
    Thread.sleep(time);
    sub.close();
  }
}
```

Test

```
java -cp build;%JBOSSCLIENT_PATH% Publisher
java -cp build;%JBOSSCLIENT_PATH% Subscriber 60000
```

Mehrere Subscriber können in verschiedenen DOS-Fenstern mit unterschiedlichen Aktivitätszeiten (im Beispiel 60 Sekunden) gestartet werden.

8.5 Dauerhafte Subscriber

Wie bereits in Kapitel 8.4 erwähnt erhält ein *Subscriber* standardmäßig nur solche Nachrichten, die an ein Topic geschickt werden, während er eine Verbindung zum MOM-Server hat. Ein *dauerhafter Subscriber* kann auch nachträglich Nachrichten abrufen, die geschickt wurden, während er inaktiv war. Hierzu muss eine Subskription eindeutig identifiziert werden können.

Zur Identifizierung einer Subskription dienen

- die Client-Identifikation der Verbindung und
- ein Name, der innerhalb dieser Client-Identifikation die Subskription eindeutig identifiziert.

Die Client-Identifikation wird mit der folgenden `TopicConnection`-Methode gesetzt, unmittelbar nachdem das `TopicConnection`-Objekt erzeugt wurde:

```
void setClientID(String clientID) throws JMSException
```

Die `TopicSession`-Methode

```
TopicSubscriber createDurableSubscriber(Topic topic, String name)
    throws JMSException
```

erzeugt einen dauerhaften Subscriber. `name` ist der oben erwähnte Subskriptionsname.

Mit der `TopicSession`-Methode

```
void unsubscribe(String name) throws JMSException
```

kann sich der Subscriber vom Server abmelden.

Die `TopicConnection`-Methode

```
String getClientID() throws JMSException
```

liefert die eindeutige Client-Identifikation der Verbindung.

Programm 8.5 demonstriert Anmeldung, Abruf von Nachrichten und Abmeldung eines dauerhaften Subscribers. ***Programm 8.5***

Die XML-Datei zur Konfiguration des Topics für JBoss muss geändert werden. Die Rolle *guest* muss das Recht für eine dauerhafte Subskription erhalten: Attribut `create="true"`.

Inhalt von *myTopic2-service.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
  <mbean code="org.jboss.mq.server.jmx.Topic"
    name="jboss.mq.destination:service=Topic,name=myTopic2">
    <depends optional-attribute-name="DestinationManager">
      jboss.mq:service=DestinationManager
    </depends>
    <depends optional-attribute-name="SecurityManager">
      jboss.mq:service=SecurityManager
    </depends>
    <attribute name="SecurityConf">
      <security>
        <role name="guest" read="true" write="true" create="true"/>
      </security>
    </attribute>
  </mbean>
</server>
```

Publisher

```
import javax.jms.*;
import javax.naming.*;

public class Publisher {
  private static final String DESTINATION = "topic/myTopic2";
  private static final String USER = "guest";
  private static final String PASSWORD = "guest";

  private TopicConnectionFactory factory;
  private Topic topic;

  public Publisher() throws NamingException, JMSException {
    Context ctx = new InitialContext();
    factory = (TopicConnectionFactory) ctx.lookup(
      "ConnectionFactory");
    topic = (Topic) ctx.lookup(DESTINATION);
  }

  public void publish(String text) throws JMSException {
    TopicConnection connection = factory.createTopicConnection(
      USER, PASSWORD);
    TopicSession session = connection.createTopicSession(
      false, Session.AUTO_ACKNOWLEDGE);
    TopicPublisher publisher = session.createPublisher(topic);

    TextMessage message = session.createTextMessage();
    message.setText(text);
    publisher.publish(message);

    publisher.close();
    session.close();
    connection.close();
  }

  public static void main(String[] args) throws Exception {
    String text = args[0];
    Publisher pub = new Publisher();
    pub.publish(text);
  }
}
```

Subscriber

```
import javax.jms.*;
import javax.naming.*;

public class Subscriber {
  private static final String DESTINATION = "topic/myTopic2";
  private static final String USER = "guest";
  private static final String PASSWORD = "guest";

  private TopicConnectionFactory factory;
  private Topic topic;
  private TopicConnection connection;
  private TopicSession session;
  private TopicSubscriber subscriber;
```

```
public Subscriber() throws NamingException, JMSException {
  Context ctx = new InitialContext();
  factory = (TopicConnectionFactory) ctx.lookup(
    "ConnectionFactory");
  topic = (Topic) ctx.lookup(DESTINATION);
}

public void subscribe(String id, String name) throws JMSException {
  connection = factory.createTopicConnection(USER, PASSWORD);
  connection.setClientID(id);
  session = connection.createTopicSession(
    false, Session.AUTO_ACKNOWLEDGE);
  subscriber = session.createDurableSubscriber(topic, name);
  subscriber.close();
  session.close();
  connection.close();
}

public void unsubscribe(String id, String name)
  throws JMSException {

  connection = factory.createTopicConnection(USER, PASSWORD);
  connection.setClientID(id);
  session = connection.createTopicSession(
    false, Session.AUTO_ACKNOWLEDGE);
  session.unsubscribe(name);
  session.close();
  connection.close();
}

public void receive(String id, String name, long timeout)
  throws JMSException {

  connection = factory.createTopicConnection(USER, PASSWORD);
  connection.setClientID(id);
  session = connection.createTopicSession(
    false, Session.AUTO_ACKNOWLEDGE);
  subscriber = session.createDurableSubscriber(topic, name);
  connection.start();

  Message message;
  while ((message = subscriber.receive(timeout)) != null) {
    if (message instanceof TextMessage) {
      TextMessage textMessage = (TextMessage) message;
      System.out.println(textMessage.getText());
    }
  }
  subscriber.close();
  session.close();
  connection.close();
}
```

```
  public static void main(String[] args) throws Exception {
    String option = args[0];
    String id = args[1];
    String name = args[2];

    Subscriber sub = new Subscriber();
    if (option.equals("subscribe")) {
      sub.subscribe(id, name);
    }
    else if (option.equals("unsubscribe")) {
      sub.unsubscribe(id, name);
    }
    else if (option.equals("receive")) {
      long timeout = Long.parseLong(args[3]);
      sub.receive(id, name, timeout);
    }
  }
}
```

Test

Anmelden:

```
java -cp build;%JBOSSCLIENT_PATH% Subscriber subscribe 100 sub1
```

Nachricht publizieren:

```
java -cp build;%JBOSSCLIENT_PATH% Publisher Hallo
```

Nachrichten abrufen:

```
java -cp build;%JBOSSCLIENT_PATH% Subscriber receive 100 sub1 5000
```

Ausgabe:

```
Hallo
```

Abmelden:

```
java -cp build;%JBOSSCLIENT_PATH% Subscriber unsubscribe 100 sub1
```

8.6 Aufgaben

1. Entwickeln Sie auf Basis des P2P-Modells einen Zeitangabe-Service, der jede Minute die aktuelle Uhrzeit übermittelt. Ein Client soll diese Zeitangabe ausgeben.

2. Entwickeln Sie eine Variante zu Programm 8.3, in der die clientspezifische Verarbeitung einer empfangenen Nachricht in einem eigenen Thread ausgeführt wird.

3. Implementieren Sie auf Basis des Pub/Sub-Modells ein einfaches Chat-Programm. Die Bedienung soll über Kommandozeilen im DOS-Fenster erfolgen. Damit Nachrichten sowohl gesendet als auch empfangen werden können, muss das Programm einen *TopicPublisher* und einen *TopicSubscriber* enthalten.

Das Projektverwaltungswerkzeug Ant

Ant ist ein Werkzeug, mit dem Quellcode zusammengestellt, bearbeitet, übersetzt und Programme ausgeführt werden können. Es ist insbesondere für die Steuerung umfangreicher Erzeugungsprozesse im Java-Umfeld geeignet.

Installation

Unter der Adresse `http://ant.apache.org` befinden sich die Seiten zum Download der Binärdistribution inklusive Dokumentation. In diesem Buch wird die Version 1.7.0 genutzt.

Die Archivdatei *apache-ant-1.7.0RC1-bin.zip* kann in einem beliebigen Verzeichnis (z.B. *C:*) entpackt werden. Anschließend müssen Umgebungsvariablen (z.B. bei Windows über *System* in der *Systemsteuerung*) gesetzt werden:

```
ANT_HOME = C:\apache-ant-1.7.0RC1

JAVA_HOME = <Java-Installationsverzeichnis>

PATH = %JAVA_HOME%\bin;%ANT_HOME%\bin;...
```

Ein Beispiel

Ant wird über eine XML-Datei *build.xml* gesteuert. Dieses so genannte *Buildfile* enthält ein *Projekt*, in dem in *Targets* die einzelnen Arbeitsschritte (*Tasks*) zusammengefasst sind.

```
<project name="Demo-Projekt" default="makeJar" basedir=".">
  <property name="version" value="1.0" />
  <property name="src" value="src"/>
  <property name="build" value="build"/>
  <property name="lib" value="lib"/>
  <property name="jarname" value="demo-${version}.jar"/>

  <target name="prepare">
    <mkdir dir="${build}"/>
    <mkdir dir="${lib}"/>
  </target>

  <target name="clean" description="Aufraeumen">
    <delete dir="${build}"/>
    <delete dir="${lib}"/>
  </target>

  <target name="compile" depends="prepare" description="Compilieren">
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="makeJar" depends="compile"
    description="Jar-Datei erzeugen">
    <jar jarfile="${lib}/demo-${version}.jar" basedir="${build}"/>
  </target>
```

```
  <target name="start" description="Anwendung starten">
    <java classname="Demo" fork="yes">
      <classpath>
        <pathelement location="${lib}/${jarname}"/>
      </classpath>
    </java>
  </target>
</project>
```

Welche Targets zur Verfügung stehen, kann durch das folgende Kommando angezeigt werden (Aufruf in dem Verzeichnis, das *build.xml* enthält):

```
ant -p
```

Ausgabe:

```
Buildfile: build.xml

Main targets:

 clean    Aufraeumen
 compile  Compilieren
 makeJar  Jar-Datei erzeugen
 start    Anwendung starten
Default target: makeJar
```

Targets können von anderen Targets abhängig sein (Attribut *depends*). So hängt z.B. das Target *compile* vom Target *prepare*, das Target *makeJar* vom Target *compile* ab.

Der Aufruf von

```
ant makeJar
```

oder auch nur `ant`, da *makeJar* das Default-Target ist (siehe *project*-Tag), führt dazu, dass zunächst die Verzeichnisse *build* und *lib* angelegt (Target *prepare*), dann die Sourcen compiliert werden (Target *compile*) und schließlich die jar-Datei erzeugt wird.

Ausgabe:

```
Buildfile: build.xml

prepare:
    [mkdir] Created dir: D:\MKJava\MKCSJava\Ant\build
    [mkdir] Created dir: D:\MKJava\MKCSJava\Ant\lib

compile:
    [javac] Compiling 1 source file to D:\MKJava\MKCSJava\Ant\build
```

```
makeJar:
      [jar] Building jar: D:\MKJava\MKCSJava\Ant\lib\demo-1.0.jar

BUILD SUCCESSFUL
Total time: 10 seconds
```

Zu Beginn der Build-Datei werden Variablen `xxx`, so genannte *Properties*, gesetzt (z.B. konkrete Verzeichnisnamen). Auf diese wird später mit `${xxx}` Bezug genommen.

"`ant start`" startet das Programm *Demo*.

"`ant clean`" stellt den Anfangszustand wieder her.

Eine vollständige Liste aller Elemente und insbesondere der zur Verfügung stehenden Tasks befindet sich in der Online-Dokumentation von *Ant*.

Programmverzeichnis

Aufgabenverzeichnis

Internet-Quellen

1. Beispielprogramme und Lösungen zu den Aufgaben

Den Zugang zu den Zusatzmaterialien finden Sie auf der Website des Verlags

`www.vieweg.de`

direkt neben den bibliographischen Angaben zu diesem Buch.

Extrahieren Sie nach dem Download alle Dateien des ZIP-Archivs unter Verwendung der relativen Pfadnamen in ein von Ihnen gewähltes Verzeichnis Ihres Rechners.

Die Distribution enthält die Java-GUI-Anwendung *SQLClient*, mit der beliebige SQL-Anweisungen für relationale Datenbanken ausgeführt werden können, sofern ein JDBC-Treiber für das jeweilige Datenbankmanagementsystem vorliegt.

2. Java Standard Edition

`http://java.sun.com/javase/`

Hier finden Sie die neueste Version zur Java Standard Edition (Java SE) für diverse Plattformen sowie die zugehörige Dokumentation. Beachten Sie die für Ihre Plattform zutreffende Installationsanweisung.

3. Java-Entwicklungsumgebungen und -Editoren

Java-Editor: `http://lernen.bildung.hessen.de/informatik/javaeditor/index.htm`

Eclipse: `http://www.eclipse.org`

JCreator: `http://www.jcreator.com`

4. Projektverwaltungswerkzeug Ant

`http://ant.apache.org`

5. MySQL Community Server, GUI-Tools, Connector/J

`http://www.mysql.de`

6. Apache Derby

`http://db.apache.org/derby/`

7. Apache XML-RPC

http://ws.apache.org/xmlrpc/

8. XML-RPC-Library für PHP

http://sourceforge.net/projects/phpxmlrpc/

9. JMS

http://java.sun.com/products/jms/

10. Webserver, Application Server

Apache HTTP Server:	http://httpd.apache.org
JBoss Application Server:	http://www.jboss.org
Apache Tomcat:	http://tomcat.apache.org

11. Apache ActiveMQ

http://www.activemq.org

12. Protokoll-Spezifikationen

UDP:	http://www.ietf.org/rfc/rfc768.txt
TCP:	http://www.ietf.org/rfc/rfc793.txt
Ports:	http://www.iana.org/assignments/port-numbers
HTTP 1.0:	http://www.ietf.org/rfc/rfc1945.txt
HTTP 1.1:	http://www.ietf.org/rfc/rfc2616.txt
MIME:	http://www.ietf.org/rfc/rfc1521.txt
Base64:	http://www.ietf.org/rfc/rfc2045.txt
Basic Authentication:	http://www.ietf.org/rfc/rfc2617.txt
XML-RPC:	http://www.xmlrpc.com

Literaturhinweise

In den Vorbemerkungen des ersten Kapitels wurden die zum Verständnis nötigen Vorkenntnisse der Leserinnen und Leser zu Java, Datenbanken, SQL usw. aufgeführt.

Die folgenden Bücher sind für eine Einführung in diese Themen bzw. eine Vertiefung gut geeignet.

1. Java-Grundlagen

- Abts, D.: Grundkurs Java. Von den Grundlagen bis zu Datenbank- und Netzanwendungen. Vieweg, 4. Auflage 2004
- Fischer, G.; Wolff von Gudenberg, J.: Programmieren in Java 1.5. Springer, 1. Auflage 2005
- Harold, E. R.: Java I/O. O' Reilly, 2. Auflage 2006
- Heinzl, S.; Mathes, M.: Middleware in Java. Vieweg, 1. Auflage 2005 (Kapitel 2: Nebenläufigkeit in Java)
- Oaks, S.; Wong, H.: Java Threads. O' Reilly, 3. Auflage 2004
- Oechsle, R.: Parallele Programmierung mit Java Threads. Fachbuchverlag Leipzig, 1. Auflage 2001
- Wolmeringer, G.; Klein, T.: Profikurs Eclipse 3. Vieweg, 2. Auflage 2006

2. Ant

- Edlich, S.; Staudemeyer, J.: Ant - kurz & gut. O' Reilly, 2. Auflage 2006
- Matzke, B.: Ant. Eine praktische Einführung in das Java-Build-Tool. Dpunkt Verlag, 2. Auflage 2005

3. Verteilte Systeme, Software-Architekturen

- Bengel, G.: Verteilte Systeme. Vieweg, 3. Auflage 2004
- Dustdar, S.; Gall, H.; Hauswirth, M.: Software-Architekturen für Verteilte Systeme. Springer, 1. Auflage 2003
- Hammerschall, U.: Verteilte Systeme und Anwendungen. Pearson Studium, 1. Auflage 2005

4. Internet-/Web-Technologien

- Avci, O.; Trittmann, R.; Mellis, W.: Web-Programmierung. Vieweg, 1. Auflage 2003

- Badach, A.; Rieger, S.; Schmauch, M.: Web-Technologien. Architekturen, Konzepte, Trends. Hanser Fachbuchverlag, 1. Auflage 2003
- Comer, D. E.: Computernetzwerke und Internets. Pearson Studium, 3. Auflage 2003
- Erlenkötter, H.: X/HTML. Flexible Webseiten von Anfang an. Rowohlt Tb., 1. Auflage 2003
- Gourley, D.; Totty, B.: HTTP. The Definitive Guide. O' Reilly, 1. Auflage 2002
- Meinel, C.; Sack, H.: WWW. Kommunikation, Internetworking, Web-Technologien. Springer, 1. Auflage 2003
- Münz, S. u. a.: SELFHTML. http://de.selfhtml.org
- Musciano, C.; Kennedy, B.: HTML und XHTML. Das umfassende Referenzwerk. O' Reilly, 4. Auflage 2006
- Niederst, J.: HTML & XHTML kurz & gut. O' Reilly, 3. Auflage 2006
- Scherff, J.: Grundkurs Computernetze. Vieweg, 1. Auflage 2006
- Wöhr, H.: Web-Technologien. Dpunkt Verlag, 1. Auflage 2004

5. XML

- Erlenkötter, H.: XML. Extensible Markup Language von Anfang an. Rowohlt Tb., 1. Auflage 2003
- Ray, E. T.: Einführung in XML. O' Reilly, 2. Auflage 2004
- Vonhoegen, H.: Einstieg in XML. Galileo Press, 3. Auflage 2005

6. Datenbanken, SQL

- Eirund, H.; Kohl, U.: Datenbanken – leicht gemacht. Teubner, 2. Auflage 2003
- Kleinschmidt, P.; Rank, C.: Relationale Datenbanksysteme. Eine praktische Einführung. Springer, 3. Auflage 2004
- Kleuker, S.: Grundkurs Datenbankentwicklung. Vieweg, 1. Auflage 2006
- Kofler, M.: MySQL 5. Einführung, Programmierung, Referenz. Addison-Wesley, 3. Auflage 2005
- Kuhlmann, G.; Müllmerstadt, F.: SQL. Der Schlüssel zu relationalen Datenbanken. Rowohlt Tb., 2004
- Schicker, E.: Datenbanken und SQL. Teubner Verlag, 3. Auflage 2000
- Steiner, R.: Grundkurs Relationale Datenbanken. Einführung in die Praxis der Datenbankentwicklung für Ausbildung, Studium und IT-Beruf. Vieweg, 5. Auflage 2003
- Throll, M.; Bartosch, O.: Einstieg in SQL. Galileo Press, 1. Auflage 2004

7. JDBC

- Dehnhardt, W.: Java und Datenbanken. Hanser Fachbuchverlag, 1. Auflage 2003
- Langner, T.; Reiberg, D.: J2EE und JBoss. Hanser Fachbuchverlag, 1. Auflage 2005 (Kapitel 4)
- Reese, G.: Database Programming with JDBC and Java. O' Reilly, 2. Auflage 2000
- Saake, G.; Sattler, K.-U.: Datenbanken und Java. Dpunkt Verlag, 2. Auflage 2003

8. Socket-Programmierung

- Harold, E. R.: Java Network Programming. O' Reilly, 3. Auflage 2004
- Heinzl, S.; Mathes, M.: Middleware in Java. Vieweg, 1. Auflage 2005 (Kapitel 4)
- Langner, T.; Reiberg, D.: J2EE und JBoss. Hanser Fachbuchverlag, 1. Auflage 2005 (Kapitel 6)

9. XML-RPC

- Saint Laurent, S.; Johnston, J.; Dumbill, E.: Programming Web Services with XML-RPC. O' Reilly, 1. Auflage 2001

10. RMI

- Bengel, G.: Verteilte Systeme. Vieweg, 3. Auflage 2004 (Kapitel 3.3.1)
- Heinzl, S.; Mathes, M.: Middleware in Java. Vieweg, 1. Auflage 2005 (Kapitel 6)
- Langner, T.; Reiberg, D.: J2EE und JBoss. Hanser Fachbuchverlag, 1. Auflage 2005 (Kapitel 7)
- Pitt, E.; McNiff, K.: java.rmi, The Remote Method Invocation Guide. Addison-Wesley Longman, 2001

11. JMS

- Bengel, G.: Verteilte Systeme. Vieweg, 3. Auflage 2004 (Kapitel 3.1.2)
- Heinzl, S.; Mathes, M.: Middleware in Java. Vieweg, 1. Auflage 2005 (Kapitel 8)
- Langner, T.; Reiberg, D.: J2EE und JBoss. Hanser Fachbuchverlag, 1. Auflage 2005 (Kapitel 21 - 24)

12. Webserver, Applikationsserver

- Fleury, M.; Stark, S.; Richards, R.: JBoss 4.0. Das offizielle Handbuch. Addison-Wesley, 2005
- Rettig, O.: Tomcat 5.5. Installation, Konfiguration, sicherer Betrieb, inkl. AJAX. Galileo Press, 2. Auflage 2007
- Richards, N.; Griffith, S.: JBoss - Das Entwicklerheft. O' Reilly, 1. Auflage 2005

Stichwortverzeichnis

N

O

P

Q

R

S

T

U

V

W

X

Z