Datenbank- und Netzwerkprogrammierung in Java

Diplomarbeit
für die Prüfung für Diplom-Volkswirte
eingereicht beim
Prüfungsausschuß für Diplom-Volkswirte
der
Wirtschaftswissenschaftliche Fakultät der
Universität Heidelberg
2003

Markus Zobeley

geboren in: Eppingen

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe verfaßt habe, und daß alle wörtlich oder sinngemäß aus Veröffentlichungen entnommenen Stellen dieser Arbeit unter der Quellenangabe einzeln kenntlich gemacht sind.

_____

Gliederung

1. Einleitung

Die weltweite Vernetzung vieler Rechner durch das Internet, ermöglicht es den Anwendern, sich von verschiedenen Seiten Informationen zu beschaffen. Die meisten dieser Zugriffe auf externe Rechner sind Datenbankzugriffe.
Mit Java gibt es eine leicht zu verstehende und zu erlernbare Programmiersprache, die es auch dem nichtprofessionellen Anwendungsprogrammierer gestattet, Datenbanken in eine eigene Webseite, über die Datenbankzugriffsschnittstelle JDBC, einzubetten.

Das Internet stellt eine Vernetzung vieler heterogener Computersysteme dar, die über TCP/IP kommunizieren. Die Plattformunabhängigkeit von Java, d.h. ein Java-Quellcode kann auf den verschiedensten Betriebssysteme ausgeführt werden, ist eine für das Internet ideale Vorraussetzung, Datenbanken und deren Anbindung für ein breites Publikum bereitzustellen.
Eine weitere Möglichkeit von Java in Bezug auf Netzwerke stellt die Remote Method Invocation (RMI) dar. Mit dieser Methode lassen sich entfernte Objekte, die auf einem anderen Rechner laufen, so handhaben, als liefen sie auf dem eigenen Rechner.

In dieser Arbeit wird im zweiten Kapitel zunächst das Augenmerk auf die Datenbanken gelenkt. Es werden die Anforderungen, die Datenbanken im allgemeinen für einen einwandfreien Betrieb benötigen, erläutert. Des weiteren ist das heute am weitest verbreiteten Datenbankmodell, das relationale Datenbankmodell, Gegenstand der Diskussion.

Im dritten Kapitel wird die Programmiersprache Java näher beleuchtet, insbesondere die Objektorientierung und die Struktur.

Kern der Betrachtungen im vierten Kapitel ist die Architektur und das Treiberkonzept von JDBC. Zum Abschluss dieses Kapitels wird der Aufbau einer JDBC-Anwendung beschrieben.

Im fünften und letzten Kapitel wird dann noch kurz am Beispiel einer Anwendung auf die Besonderheiten von RMI eingegangen.

2. Datenbanken

2.1. Einleitung

In diesem Kapitel sind die verschiedenen Aspekte des sehr komplexen Themas der Datenbanken Betrachtungsgegenstand.

Zuerst stelle wird das Drei-Ebenen-Modell nach ANSI vorgestellt, das ein grundlegendes theoretisches Prinzip zum Erreichen von Datenunbhängigkeit in einem Datenbanksystem darstellt.

Im zweiten Punkt wird auf die allgemeinen Anforderungen einer Datenbank eingegangen und besonders die Organisation des Mehrbenutzerbetriebes betrachtet.

Anschließend wird ein sehr wichtiges Instrument zur Entwicklung eines Datenbanksystems dargestellt, das Entity-Relationship-Modell (ERM).

Danach wird das zur Zeit am weitest verbreiteten Datenbankmodell, das relationale Datenbankmodell, erläutert.

Die Abfragesprache SQL (Structured Query Language) rundet dieses Kapitel ab.

2.2. Das Drei-Ebenen-Modell von ANSI

Das American National Standardization Institute definierte ein Standardschema, mit dem die Trennung der Programmlogik von der tatsächlichen physischen Abspeicherung möglich ist. Folgende Abbildung stellt dieses Drei-Ebenen-Modell dar.

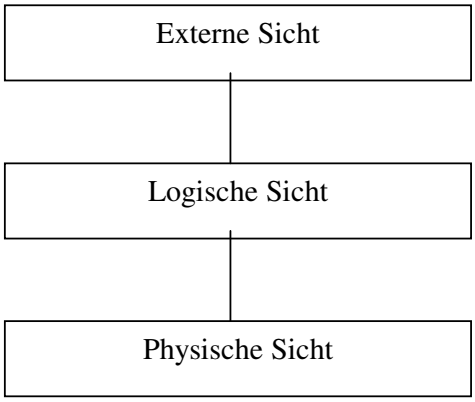


Abb. 1.: Drei-Ebenen-Modell nach ANSI[1]

Im folgendem werden die verschiedenen Ebenen kurz erläutert[2]:

Externe Sicht

Umfasst die Sichten (Views) der Endbenutzer auf die Datenbank. Der Endbenutzer muss nichts über die Art und Weise der Datenspeicherung wissen, noch wo diese Daten gespeichert sind. Der Endbenutzer richtet seine Abfrage an das Datenbankmanagementsystem (DBMS). Dieses liefert dann die gewünschten Informationen an den Benutzer[3].

[1] Vgl. Rautenberg,Schulze: Informatik für Wirtschaftswissenschaftler und Wirtschaftsinformatiker, S.124f
[2] Vgl. Rautenberg,Schulze: Informatik für Wirtschaftswissenschaftler und Wirtschaftsinformatiker, S.125
[3] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 20

Logische Sicht

Dies ist die Schicht des Datenbankprogrammierers. Dort werden die Metadaten des Datenbanksystems festgelegt und abgespeichert. Des weiteren werden auf dieser Ebene Informationen über die Zugriffspfade und -rechte, sowie der Dateiorganisation festgehalten[4]. Der Datenbankadministrator, der auf dieser Ebene die Zugriffsrechte der Benutzer auf die Datenbank verwaltet und die Datenbank pflegt, sollte dafür Sorge tragen, dass die Datenbank die an sie gerichtete Anforderungen möglichst effizient und sicher erfüllt.

Physische Sicht

Auf dieser Ebene werden die Daten nach dem vom Datenbankprogrammierer auf der logischen Ebene festgelegten Schema effizient, d.h. schnell abfragbar, gespeichert[5].

2.3. Allgemeine Anforderungen an Datenbanken

Datenbanken sollten eine Reihe von Anforderungen erfüllen, damit der Betrieb reibungslos vonstatten geht. Nun werden einige wichtige Eigenschaften von Datenbanken erläutert, auf den Mehrbenutzerbetrieb wird näher eingegangen.

2.3.1.Redundanzfreiheit

Redundanzfreiheit einer Datenbank bedeutet, dass jedes Datenelement nur einmal abgespeichert wird[6]. Damit soll verhindert werden, dass ein Datenelement in verschiedenen Versionen an verschiedenen Stellen abgespeichert wird. Da aber eine völlig redundanzfreie Datenbank bei einer komplexen Auswertung Nachteile in der Performance im Vergleich zu

[4] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 19
[5] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 20
[6] Vgl. Disterer, Fels, Hausotter : Taschenbuch der Wirtschaftsinformatik, S. 209

nicht völlige redundanzfreien Datenbanken haben kann, ist es nicht immer sinnvoll auf völlige Redundanzfreiheit zu bestehen.

2.3.2. Unabhängigkeit

Unter Unabhängigkeit versteht man die Unabhängigkeit der Daten von den Endbenutzern. Der Endbenutzer richtet seine Abfrage nicht direkt an die Datenbank, sondern an das Datenbankmanagementsystem (DBMS). Das wiederum bearbeitet diese Anfrage, indem das DBMS die benötigten Daten aus der Datenbank ausliest und sie dem Endbenutzer zur Verfügung stellt. Ein solches System bietet das oben beschriebene ANSI-Modell[7].

2.3.3. Integrität

Die zentral gespeicherten Daten sollten sich, hinsichtlich des Datenschutzes und der Datensicherheit, besonderer Aufmerksamkeit erfreuen. Der Datenschutz ist in der heutigen Zeit ein Thema, das ständig in der Diskussion steht. Datenschutz bedeutet, dass eine Datenbank gegenüber illegalen Zugriffen geschützt wird. Dies zu gewährleisten ist Aufgabe des Datenbankadministrators.
Um Datensicherheit zu erreichen, müssen Maßnahmen ergriffen werden, die die physische Zerstörung der Daten durch Gerätefehler, äußere Einflüsse, Zerstörung von Daten durch Programmabbrüche, Eingabefehler oder Fehler, die durch simultanen Zugriff entstehen, verhindern[8].

Des weiteren sollte eine Datenbank benutzerfreundlich, also leicht und verständlich zu bedienen sein und sie sollte die an sie gerichtete Leistungsmerkmale (kurze Antwortzeiten, Ausfallsicherheit usw.) erfüllen.

[7] Vgl. Heuer/Saake/ Sattler : Datenbanken kompakt, S.55
[8] Vgl. Rautenstrauch/Schulze: Informatik für Wirtschaftswissenschaftler und Wirtschaftsinformatiker, S.201ff

2.3.4. Mehrbenutzerbetrieb

Der Zugriff auf Datenbanken ist ein zentrales Thema dieser Diplomarbeit. Aus diesem Grunde wird nun die Organisation des Mehrbenutzerbetriebes, d.h. wenn mehrere User gleichzeitig auf eine Datenbank zugreifen, näher beleuchtet.

Jede Anfrage eines Benutzers an eine Datenbank stellt eine Transaktion.
Transaktionen sind unteilbare Operationen auf Datenbanken. Die Verwaltung dieser Transaktionen ist eine zentrale Aufgabe der Datenbankverwaltung.

2.3.4.1. Das ACID-Prinzip[9]

Das ACID-Prinzip ist grundlegend für alle Transaktionen und garantiert dem User eine konsistente Datenbank. Nachfolgend eine Erläuterung der einzelnen Aspekte dieses Prinzips.

(A)tomicity (atomar)
Transaktionen sind atomar, d.h. sie werden als Ganzes abgeschlossen oder aber, sollte es nicht möglich sein, die Transaktion komplett auszuführen, werden sie komplett zurückgesetzt. In diesem Fall sind keinerlei Auswirkungen auf die Daten der Datenbank zu bemerken.

(C)onsistency (Konsistenz)

Die Konsistenz einer Datenbank ist dann gegeben, wenn sich eine Datenbank nach Ausführung einer Transaktion in einem konsistenten Zustand befindet. Dabei ist es egal, ob eine Transaktion ausgeführt (COMMIT) oder abgebrochen (ROLLBACK) wurde.
Die Datenbank wird durch eine Transaktion von einem konsistenten, also widerspruchfreien, in einen anderen konsistenten Zustand überführt.

[9] Vgl. Rautenstrauch/Schulze: Informatik für Wirtschaftswissenschaftler und Wirtschaftsinformatiker, S.126 und Fink, Schneidereit, Voß: Grundlagen der Wirtschaftsinformatik, S. 137f

(I)solation

Eine Transaktion sollte immer isoliert ablaufen, dann ist der erfolgreiche Abschluss einer Transaktion unabhängig von etwaigen gleichzeitig laufenden Transaktionen. Das Isolations-Prinzip soll unerwünschte „Seiteneffekte“ parallel laufender Transaktionen verhindern.

(D)uration (Dauerhaftigkeit)

Eine abgeschlossene Transaktion wird dauerhaft festgeschrieben. Wurde eine Transaktion erfolgreich abgeschlossen, so bleibt die Wirkung dieser Transaktion bestehen, bis sie von einer anderen Transaktion ausdrücklich widerrufen wird. Dieses Prinzip wird auch Persistenz einer Datenbank genannt.

2.3.4.2. Serialisierbarkeit

Um den Mehrbenutzerbetrieb zu gestalten und ein für jeden User unabhängiges Arbeiten auf einer Datenbank zu ermöglichen müssen Transaktionen serialisierbar sein.
Grundsätzlich gibt es zwei verschiedene Methoden um Transaktionen zu serialisieren.
Zum einen gibt es pessimistische Verfahren, die keine Konflikte bei parallelen Transaktionen zulassen und zum anderen optimistische Verfahren, die Konflikte zulassen und diese durch Zurücksetzen lösen. Eine genauere Betrachtung folgt.

2.3.4.2.1. Pessimistisches Verfahren

Das exklusive Sperren eines Objektes ist ein pessimistisches Verfahren zur Serializierbarkeit von Transaktionen. Bei dieser Methode wird ein Objekt exklusiv von einem Benutzer gesperrt und erst wieder für andere Benutzer verwendbar, wenn es freigegeben wurde.

Eine Erweiterung des exklusiven Sperrens ist das Zwei-Phasen-Sperrprotokoll. Falls ein User aus einer Datenbank zwei oder mehrere Objekte bearbeiten möchte, so kann der User die benötigten Objekte sperren (Wachstumsphase) und nach der Bearbeitung wieder freigeben (Schrumpfungsphase). Dabei ist es dem User nicht möglich, unmittelbar nach der Freigabe eines Objektes, dieses wieder für sich zu sperren. Das Sperren und Freigeben der verschiedenen Objekten kann entweder zu Beginn, bzw. zum Ende einer Transaktion, oder aber auch sukzessive erfolgen, d.h. ein Objekt wird erst gesperrt wenn es benötigt wird und freigegeben, wenn es nicht mehr benötigt wird[10].

Durch dieses Verfahren sind die Transaktionen serialisiert, da kein User auf Objekte zugreifen kann, an denen gerade gearbeitet wird.
Der Nachteil dieses Verfahrens ist die unter Umständen lange Wartezeit auf ein Objekt. Deshalb wurden andere Verfahren entwickelt.

2.3.4.2.2. Optimistische Verfahren

Unter der Annahme, dass Konflikte beim Zugriff auf Datenbanken nur sehr selten vorkommen wird bei diesem Verfahren auf das Setzen von Sperren verzichtet[11].
Bei optimistischen Verfahren durchlaufen Transaktionen drei Phasen: die Lesephase, die Validierungsphase und die Schreibphase.

In der Lesephase können alle Benutzer auf die benötigen Objekte zugreifen und in ihren Speicher lesen.
In der Validierungsphase wird durch das System geprüft, ob eine Änderung durch eine Transaktion an einem Objekt möglich ist, d.h. ob ein anderer User zur Zeit an diesem Objekt Arbeiten vornimmt. Ist dies der Fall, so prüft das Datenbanksystem, wer als erster auf das Objekt zugegriffen hat und setzt dann den anderen User zurück.
Die Schreibphase schließlich besteht aus dem Vollenden einer Transaktion, die erfolgreich die Validierungsphase absolviert hat. In dieser Phase werden die Änderungen an einem Objekt der Datenbank durchgeführt und für alle anderen Benutzer sichtbar.

[10] Vgl. Disterer, Fels, Hausotter : Taschenbuch der Wirtschaftsinformatik, S. 272ff
[11] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 237

Ein Vorteil dieses Verfahrens im Vergleich zu dem pessimistischen Verfahren ist, dass es während der Schreibphase keine Konflikte zwischen den Transaktionen gibt. Diese können erst in der Validierungsphase auftreten und werden dort durch Zurücksetzen gelöst.

2.4. Das Entity-Relationship-Modell (ERM)

Das ERM geht auf einen grundlegenden Artikel von P.P.Chen aus dem Jahre 1976 zurück. Seit dieser Zeit hat sich dieses Modell faktisch als Standardmodell zur Modellierung für Datenbankentwürfe entwickelt. Aus einem ER-Modell lassen sich relationale Datenbanksysteme erstellen, die im nächsten Abschnitt behandelt werden In diesem Abschnitt werden die grundlegenden Begrifflichkeiten des ERM erläutert.

2.4.1. Begriffe des ER-Modelles

Entity (Entität)

„Eine Entity ist eine Objekt der realen Welt oder der Vorstellungswelt, über das Informationen zu speichern sind [...].“[12] Beispiele für Entitäten wären Maschinen, Mitarbeiter usw.. Entitäten werden im ERM als Rechtecke dargestellt.

Entitätsmenge

Eine Entitätsmenge stellt ein Menge von Entitäten des gleichen Typs dar, z.B. alle Mitarbeiter eines Betriebes etc.[13].

Attribute

Die Eigenschaften einer Entität bezeichnet man als Attribut. Als Beispiel seien die Attribute der Entität Mitarbeiter Name,Alter,Adresse erwähnt. Attribute werden als Kreise im ER-Modell gezeichnet[14].

[12] Vgl. Heuer/Saake/ Sattler : Datenbanken kompakt, S.55
[13] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 86
[14] Vgl. Heuer/Saake/ Sattler : Datenbanken kompakt, S.55

Domäne

Der Wertebereich eines Attributes wird Domäne genannt. Bsp.: Die Domäne des Attributs Alters wäre zum Beispiel {0,1,2,..,120}[15].

Schlüssel

Das Schlüsselattribut dient zur eindeutigen Ermittlung einer Entität. Es muss eindeutig sein, d.h. in einer Entitätsmenge dürfen keine zwei Entitäten mit dem gleichen Wert des Schlüsselattributes vorkommen[16].

Relationship

Eine Relationship bezeichnet eine Beziehung zwischen Entitäten. Beziehungen werden im ER-Modell als Rauten gekennzeichnet[17].

Zur Verdeutlichung der hier erläuterten Begriffe folgt im nächsten Abschnitt ein einfaches Entity-Relationship-Modell Schema.

[15] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 87
[16] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 89
[17] Vgl. Heuer/Saake/ Sattler : Datenbanken kompakt, S.55

2.4.2. Ein einfaches ER-Modell

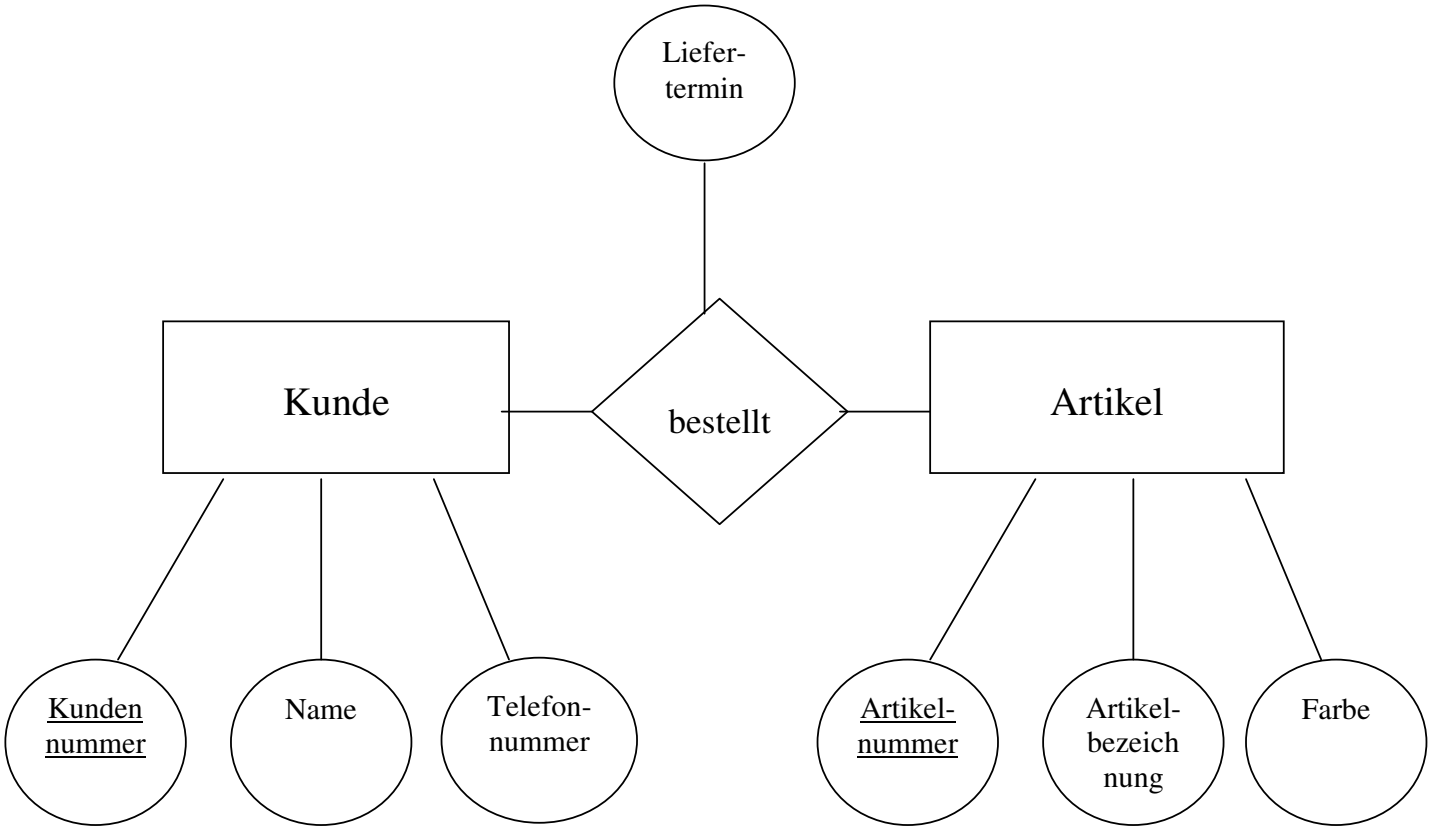


Abb 2.: Ein einfaches ER-Modell[18]

Dieses einfache Schema besteht aus den Entitäten ’Kunde’ und ’Artikel’. Die Beziehung dieser Entitäten besteht über die Bestellung, ein Kunde bestellt einen Artikel.
Die Attribute der einzelnen Entitäten lassen sich in den Kreisen ablesen, die unterstrichenen Attribute sind die Schlüssel der jeweiligen Entität.
Des weiteren verdeutlicht dieses Beispiel, dass auch die Relationship eigene Attribute haben kann, in diesem Beispiel wäre es der Liefertermin.

[18] selbsterstellt, in Anlehnung an: Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.28

2.4.3. Die Assoziationstypen

Wie aus Abbildung 2 hervorgeht, gibt es verschiedene Möglichkeiten der funktionalen Verbindungen zwischen Entitäten. Ein Kunde kann keinen, einen oder auch mehrere Artikel bestellen, während ein Artikel von mehreren Kunden gekauft werden kann (wenn mehrere auf Lager vorhanden sind).

Deshalb kann man im ER-Modell verschiedene Assoziationstypen unterscheiden. Es gibt grundsätzlich vier verschieden Typen[19].

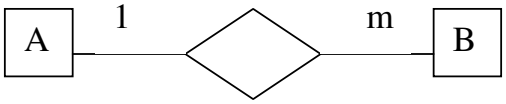
Die einfache Assoziation (Typ 1): Genau eine Entität aus einer Entitätsmenge steht in Beziehung.

Die konditionelle Assoziation (Typ c): Keine oder eine Entität einer Entiätsmenge steht in Beziehung.

Der mehrfache Typ (Typ m): Mehrere Entitäten aus einer Entitätsmenge stehen in Beziehung.

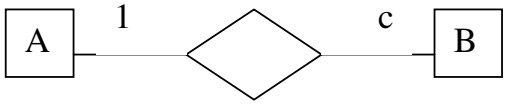
Der mehrfach-konditionelle Typ (Typ mc): Keine, eine oder mehrere Entitäten einer Entitätsmenge stehen in Beziehung.

Um die oben beschriebenen Assoziationstypen besser darzustellen folgen nun ein paar Beispiele:

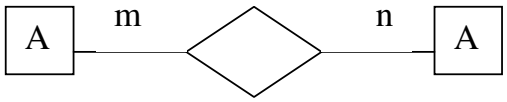


Eine 1:M-Beziehung: Entitätsmenge A sind Flüge und Entitätsmenge B sind Passagiere. Jeder Flug ist mit mehreren (Typ m) Passagieren besetzt, aber ein Passagier kann nur in einem Flugzeug gleichzeitig sitzen.

[19] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 92



In diesem Fall könne man sich vorstellen, dass A die Abteilungen einer Firma sind, B sind die Mitarbeiter und die beiden Entitätsmengen stehen über den Abteilungsleiter in Beziehung.
So hat jede Abteilung genau einen Abteilungsleiter, aber nicht jeder Mitarbeiter ist ein Abteilungsleiter.



Jede Entität aus der Entitätsmenge A hat eine Beziehung zu mehreren Entitäten der Entitätsmenge B. Bsp.: Professoren und Studenten. Jeder Professor hat mehrere Studenten in seiner Vorlesung und jeder Student besucht mehrere Veranstaltungen (hoffentlich).

2.5. Das relationale Datenbankmodell

Das relationale Datenbankmodell geht auf den Artikel „A Relational Model of Data for Large Shared Data Banks“ von E.F.Codds aus dem Jahre 1970 zurück. In diesem Modell werden die Daten in Relationen (Tabellen) verwaltet und gespeichert. Da sich dieser Modellierungsansatz in der Praxis durchgesetzt hat und in sehr vielen Unternehmensdatenbanken Anwendung findet, gehe ich in diesem Unterkapitel ausführlicher auf das Modell der relationalen Datenbanken ein.

2.5.1. Begriffe des relationalen Datenbanksystems

Relation

Eine Relation ist eine zweidimensionale Tabelle, die aus Zeilen, den Tupeln, und aus Spalten, den Attributen, besteht. Formal betrachtet ist eine Relation die Teilmenge aus dem kartesischen Produkt der Domänen aller Attribute[20].
Jede Relation hat einen eindeutigen Namen.

Tupel

Ein Tupel bezeichnet eine Zeile einer Relation. Tupel werden häufig auch Datensatz genannt. Die Anzahl der Tupel einer Relation nennt man Kardinalität, wobei die Anordnung der Tupel in einer Tabelle ohne Bedeutung ist, da jeder Tupel eindeutig über den Wert des Primärschlüsselattributes zu identifizieren ist[21].

Attribut

Die Spalten einer Relation sind die Attribute. Ein Attribut in einer Relation muss einen eindeutigen Namen haben, da die Attribute über ihren Attributnamen angesprochen werden.
Der Wertebereich eines Attributs wird Domäne genannt[22].

Primärschlüssel

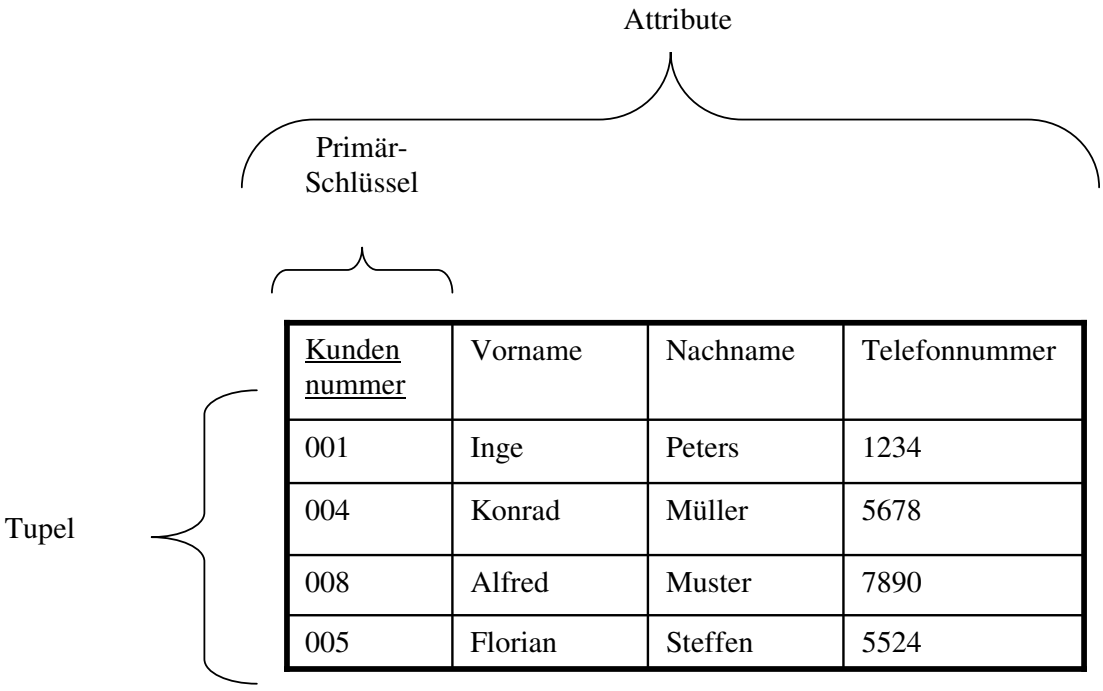
Der Primärschlüssel ist das Attribut, über das jeder einzelne Tupel identifiziert werden kann. Deshalb ist es notwendig, das jedes Tupel an der Stelle des Schlüsselattributes einen anderen Wert hat, denn sonst ist die eindeutige Unterscheidbarkeit der Tupel nicht mehr gewährleistet. Neben einem einfachen Primärschlüssel gibt es kombinierte Schlüssel, die aus mehreren Attributen bestehen und durch eine minimale Attributkombination gekennzeichnet sind. Bei einer minimalen Attributkombination kann kein Attribut weggelassen werden, ohne dass die Identifizierung der Tupel verloren geht[23].

---

[20] Vgl. Rautenstrauch/Schulze : Informatik für Wirtschaftswissenschaftler und Wirtschaftsinformatiker, S.129,130
[21] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 35f
[22] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 32f
[23] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 40f

Der Grad einer Relation entspricht der Anzahl der Attribute, über die die Relation gebildet wird.

Hier ein einfaches Beispiel einer selbsterstellten Relation:



Kundennummer	Vorname	Nachname	Telefonnummer
001	Inge	Peters	1234
004	Konrad	Müller	5678
008	Alfred	Muster	7890
005	Florian	Steffen	5524

Der Grad dieser Relation wäre vier.

2.5.2. Die Normalisierung

Um ein konsistentes und redundanzfreies Datenbanksystem zu erreichen wird die Normalisierung angewandt, deren einzelne Schritte über die Normalformen laufen. Es gibt schon über 20 Normalformen, in dieser Arbeit werden nur die ersten drei Normalformen erläutert.

2.5.2.1. Erste Normalform

Eine Relation befindet sich in der ersten Normalform, wenn die Wertebereiche der Attribute atomar sind[24].

Bsp.: Tabelle „Mitarbeiter“, die nicht in der 1.Normalform ist

MitarbeiterNr.	Name	Projektnummer	Projektbezeichnung
12	Schulze	1	Marketingstrategien
15,24	Meyer	2	Kostensenkung
18	Arnold	2	Kostensenkung
20	Schmidt	1	Marketingstrategien

Annmerkung: MitarbeiterNr. und Projektnummer sind ein zusammengesetzter Schlüssel

Dieselbe Tabelle in der 1.NF:

MitarbeiterNr.	Name	Projektnummer	Projektbezeichnung
12	Schulze	1	Marketingstrategien
15	Meyer	2	Kostensenkung
24	Meyer	2	Kostensenkung
18	Arnold	2	Kostensenkung
20	Schmidt	1	Marketingstrategien

Diese Tabelle befindet sich nun in der ersten Normalform, d.h. alle Attributwerte sind atomar. Dennoch gibt es noch Redundanzen in dieser Tabelle.

2.5.2.2. Zweite Normalform

Eine Tabelle befindet sich in der zweiten Normalform, wenn alle Nichtschlüsselelemente voll funktional vom Gesamtschlüssel abhängen.

Ein Nichtschlüsselelement ist dann voll funkional vom Gesamtschlüssel abhängig, wenn es nicht von einem Teilschlüssel abhängig ist, sondern ausschließlich von dem Gesamtschlüssel[25].

[24] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 76
[25] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 76f

Die Tabelle aus dem vorhergehendem Punkt wäre nicht in der zweiten Normalform, weil das Attribut Projektbezeichnung nicht voll funktional abhängig ist vom Gesamtschlüssel (MitarbeiterNr,Projektnummer), sondern eine funktionale Abhängigkeit zum Teilschlüssel Projektnummer aufweist.

Um diese Beispieltabelle in die zweite Normalform zu bringen, muss eine neue Tabelle für die Projektnummer und Projektbezeichnung erstellt werden. Von dieser neu zu erstellenden Tabelle wird nun ein Fremdschlüssel und die ursprüngliche Tabelle referenziert.
Ein Fremdschlüssel muss immer Primärschlüssel in der referenzierten Tabelle sein.

Bsp: So muss die Tabelle aus Kapitel 2.5.2.1. in zwei Tabellen aufgeteilt werden, damit die zweite Normalform erreicht wird:

Tabelle Mitarbeiter

MitarbeiterNr.	Name	Projektnr.
12	Schulze	1
15	Meyer	2
24	Meyer	2
18	Arnold	2
20	Schmidt	1

Neue Tabelle Projekte

Projektnummer	Projektbezeichnung
1	Marketingstrategien
2	Kostensenkung

Annmerkung: In der ursprünglichen Tabelle Mitarbeiter wurde nun die Zeile Projektnr. aufgenommen, die ein Fremdschlüssel ist und eine Referenz auf die Tabelle Projekt darstellt.

2.5.2.3. Dritte Normalform

Eine Relation befindet sich genau dann in der dritten Normalform, wenn es keine transitiven Abhängigkeiten gibt, d.h. es gibt keine Abhängigkeiten zwischen Nichtschlüsselattributen, sondern jedes Nichtschlüsselattribut hängt nur vom Schlüsselattribut ab[26].

Bsp.: Eine neue Tabelle „Mitarbeiter“, die nicht in der dritten Normalform ist:

MitarbeiterNr.	Name	Telefonnummer	PLZ	Wohnort
23	Schulze	45582	74080	Heilbronn
43	Schmidt	86545	74080	Heilbronn
45	Meier	78655	69117	Heidelberg
56	Schwan	55426	69117	Heidelberg

Es besteht eine transitive Abhängigkeit zwischen dem Nichtschlüsselattribut PLZ und dem Wohnort. Um diese Tabelle in die dritte Normalform zu bringen, ist analog zur zweiten Normalform vorzugehen.

Tabelle Mitarbeiter

MitarbeiterNr.	Name	Telefonnummer	PLZ
23	Schulze	45582	74080
43	Schmidt	86545	74080
45	Meier	78655	69117
56	Schwan	55426	69117

Tabelle Wohnort

PLZ	Wohnort
74080	Heilbronn
69117	Heidelberg

In der Tabelle „Wohnort“ ist PLZ nun Primärschlüssel, in der Tabelle „Mitarbeiter“ Fremdschlüssel. Damit sind beide Tabellen in der dritten Normalform.

[26] Vgl. Matthiessen, G.; Unterstein,M.: Relationale Datenbanken und SQL, S. 78

2.5.3. Strukturelle Integritätsbedingungen

Um den Betrieb eines relationalen Datenbanksystems zu gewährleisten müssen grundlegende Integritätsbedingungen erfüllt sein.
Um eine redundanzfreie Datenbank zu erhalten wurde im oberen Kapitel die Normalisierung vorgestellt. In diesem Kapitel werden in gegebener Kürze die Maßnahmen vorgestellt, die die strukturelle Integrität einer Datenbank erhalten sollen[27].

Eindeutigkeitsbedingung

Jeder Tupel einer Tabelle kann eindeutig über einen Primärschlüssel ausgegeben werden.

Wertebereichsbedingung

Der Wertebereich eines Attributes muss so gewählt werden, dass kein Element den Wertebereich verlässt. Die Datenwerte können also nur den vorgegebenen Wertebereich annehmen.

Referentielle Integrität

Um die referentielle Integrität eines Datenbanksystems zu erreichen muss zu jedem Fremdschlüssel in einer Tabelle ein zugehöriger Primärschlüssel in einer anderen Tabelle vorhanden sein. Durch die referentielle Integrität wird sichergestellt, dass keine unzulässigen Datensätze in eine Tabelle eingefügt werden, welche die Fremdschlüsselbedingung verletzen.

[27] Vgl. Rautenstrauch/Schulze: Informatik für Wirtschaftswissenschaftler und Wirtschaftsinformatiker, S.127f

2.6. SQL

Die Structured Query Language (SQL) bildet de facto eine durch ihre weite Verbreitung einen Standard bei den Abfragesprachen. Allerdings ist SQL keine reine Abfragesprache, sondern dient des weiteren als Data Manipulation Language (DML), d.h. mit SQL lassen sich Daten verändern und als Data Description Language (DDL), d.h. SQL kann Metadaten definieren.
Da die Java Database Connectivity (JDBC), die Methode mit der Datenbanken in Java eingebunden werden (s.Kap.4), mit SQL-Statements arbeitet, wird in diesem Kapitel die einzelnen Aspekte dieser Sprache erläutert, ohne den Befehlssatz erschöpfend darzustellen[28].

2.6.1. Der SELECT-Befehl

Der grundlegende Befehl zur Datenbankabfrage über SQL ist der SELECT-Befehl.
Der vollständige abstrakte Befehl lautet[29]:

```
SELECT DISTINCT (columns)
  FROM (table)
  WHERE (search_conditions)
  GROUP BY (columns)  HAVING (search_conditions)
  ORDER BY (sort_orders)
```

In der ersten Zeile werden die abzufragenden Attribute (columns) eingegeben. Der DISTINCT-Befehl ist optional, er dient dazu doppelte Datensätze bei der Ergebnismenge auszuschließen[30].

In der zweiten Zeile wird die Tabelle eingegeben, aus der Daten geliefert werden sollen.
Die nächste Zeile dient der Eingabe möglicher Suchkriterien, die die Abfrage einengen sollen.
Die beiden letzten Zeilen dienen zur Aufbereitung des Suchergebnisses nach den Wünschen des Users und werden hier, aufgrund ihrer Vielfältigkeit in der Ausprägung, nicht näher erklärt.

---

[28] Um die Möglichkeiten von SQL und den Befehlssatz genauer zu betrachten empfehle ich das Buch „SQL lernen“ von Michael Ebner

[29] Vgl . Ebner, Michael : SQL lernen, S.43

[30] Vgl . Ebner, Michael : SQL lernen, S.46

Um ein Beispiel eines Select-Befehles zu geben, wird hier Bezug auf die Tabelle „Mitarbeiter“ aus dem Kapitel 2.5.3. genommen, wobei der Name und die Telefonnummer des Mitarbeiters mit der Mitabeiternummer 43 abgefragt wird:

Der Befehl: SELECT (Name,Telefonnummer) FROM (Mitarbeiter)
WHERE (MitarbeiterNr = 43)

würde als Ergebnis liefern:

Schmidt	86545

2.6.2. Die DML

Die Befehle der Data Manipulation Language erstrecken sich über das Einfügen, das Ändern und das Löschen von Datensätzen in einer Datenbank. Da diese Operationen über SQL-Statements sehr umständlich einzugeben wären, wird dies im Regelfall über Datenbankapplikationen erledigt[31].
Das Einfügen eines Datensatzes über ein einfaches Statement birgt außerdem die Gefahr, dass referentielle Integritätsbedingungen (Fremdschlüsselverletzung) oder der Wertebereich eines Attributes überschritten wird. Dies kann durch eine entsprechende programmierte Datenbankapplikation verhindert werden.

2.6.3. Die DDL

Mit der Data Description Language werden die Metadaten eines Datenbanksystems definiert. Diese Daten bestehen aus den Attributen, und deren Wertebereich, einer Tabelle.
Dabei ist sorgfältig darauf zu achten, dass die Tabellen und Domänen alle Integritätsanforderungen erfüllen.

[31] Vgl . Ebner, Michael : SQL lernen, S.85

3. Java

3.1. Einleitende Worte

Die Firma Sun Microsystems entwickelte ab 1993 die Programmiersprache Java. Ziel war es, eine einfach zu lernende, höhere Programmiersprache bereitzustellen. Mit dem zu dieser Zeit einsetzenden Boom des Internets, das verschiedenartige Computersysteme miteinander verbindet, war es nötig, eine plattformunabhängige Sprache zu entwickeln[32].

Um diese Plattformunabhängigkeit zu erreichen wird der Quellcode eines Javaprogrammes durch den Java-Compiler in einen sog. Bytecode kompiliert. Der Bytecode liegt im Binärformat vor und kann deshalb von den unterschiedlichsten Computersystemen ausgeführt werden. Die Ausführung eines Javaprogrammes übernimmt der Java-Interpreter[33].

Bei der Entwicklung von Java ging Sun Microsystems von der bis dahin am weitest verbreiteten objektorientierten Sprache, C++, aus. Um den Einstieg auch für Anfänger zu vereinfachen, wurde besonders darauf geachtet, die Speicherverwaltung deutlich zu vereinfachen. War es in C++ noch nötig, die Speicherverwaltung mit Zeigern vorzunehmen, so übernimmt Java diese Aufgabe nun selber mit der automatischen Speicherverwaltung[34].

In diesem Kapitel soll auf die grundlegenden Konzepte der Objektorientierung und deren Ausprägungen in Java eingehen.
Der zweite Teil erläutet die Struktur von Java, also deren Datentypen und Programmflussteuerung.
Eine weiterführende Beschreibung von Java, insbesondere konkrete Programmierbeispiele, würden den Rahmen dieser Arbeit sprengen und werden deshalb nicht vorgenommen.

---

[32] Vgl. Wolff, Christian: Einführung in Java, S.13f

[33] Vgl.: Louis, D., Müller, P. : Jetzt lerne ich Java, S.26f

[34] Vgl. Wolff, Christian: Einführung in Java, S. 14

3.2. Objektorientierung in Java

3.2.1. Klassen

Eine Java-Applikation besteht im Grunde aus einer oder mehreren Klassen. Jede Klasse hat bestimmte Eigenschaften und Methoden, die innerhalb des Klassenrumpfes definiert sind.

Neben der Möglichkeit, eine Klasse selbst zu schreiben, kann man auf die umfangreiche Klassenbibliothek des JDK (Java Development Toolkit) zurückgreifen. Dieses Toolkit besteht in der Version JDK 1.2. aus 60 Paketen mit 1781 Klassen bzw. Schnittstellen. Ein Paket ist eine Gruppe von Klassen, die logisch zusammengehören. Deswegen ist es nicht nötig, jede Klasse, die in einem selbstgeschriebenen Programm gebraucht wird, einzeln zu importieren.
Eine Klasse wird mit dem Schlüsselwort *class* angelegt. (Schlüsselwörter dürfen nicht vom Programmierer zur Bezeichnung von Variablen etc. verwendet werden)
Zusätzlich lassen sich in dieser Zeile einen Zugriffsmodifizierer vereinbaren.
Der Zugriffsmodifizierer *public* legt fest, das eine Klasse auch außerhalb eines Paketes sichtbar ist. Lautet der Befehl *private class*, dann ist eine Klasse nur innerhalb eines Paketes sichtbar und kann nur dort verwendet werden[35].

Eine typische Klassendeklaration sähe dann folgendermaßen aus:

```
public class meineKlasse
```

Um mit einer Klasse arbeiten zu können, muss diese instantiiert werden, damit die Klasse zu einem Objekt wird. Die Bildung einer Instanz einer Klasse erfolgt über den Konstruktor.

[35] Vgl. Wolff, Christian: Einführung in Java, S. 113

3.2.1.1. Eigenschaften von Klassen

Die Eigenschaften einer Klasse werden i.d.R. unmittelbar nach dem Klassenkopf, der aus der Klassendeklaration besteht definiert. Eigenschaften einer Klasse sind die zu vereinbarenden Variablen (sowohl einfache Datentypen als auch Referenztypen , s.a. weiter unten). Einfache Datentypen werden mit dem Datentyp (z.B. Integer), dem Bezeichner und der Initialisierung bekannt gemacht, wobei die Initialisierung auch später erfolgen kann[36].

Beispiel einer Variablendeklaration mit Initialisierung:

```
int i = 5;
```

Diese Variablen werden Klassenvariablen genannt und stehen sämtlichen Methoden dieser Klasse zur Verfügung.

3.2.1.2. Methoden

Die Methoden einer Klasse definieren die Funktionalität der Klasse. In den Methoden wird festgelegt, welche Operationen mit dem Objekt einer Klasse möglich sind[37].
Die Methodendeklaration erfolgt nach folgendem Muster:

```
Zugriffsmodifizierer Rückgabewert  Methodenname ( ÜbergabeParameter );
```

Der Zugriffsmodifizierer legt die Sichtbarkeit der Methode fest. Es gibt drei Arten :

Public: Die Methode ist überall dort erreichbar, wo auch die Klasse erreichbar ist.

Protected: Die Methode ist innerhalb eines Packetes sichtbar, und den daraus abgeleiteten Klassen.

Private: Methode ist nur innerhalb der Klasse erreichbar , in der sie implementiert ist.[38]

---

[36] Vgl. Wolff, Christian: Einführung in Java, S. 115
[37] Vgl. Wolff, Christian: Einführung in Java, S. 116
[38] Vgl. Flanagan, D.: Java in a nutshell, S. 242

Jede Methode muss einen Rückgabewert haben. Ist dies nicht der Fall, so steht an der Stelle des Rückgabewertes das Schlüsselwort *void.*

Der Punkt Methodenname spricht für sich.

Die ÜbergabeParameter sind diejenigen Parameter, die bei Aufruf der Methode dieser übergeben werden[39].

Bsp einer Methode:

```
public int quadrieren (int ersteZahl,int zweiteZahl) {
                        return ersteZahl * zweiteZahl
                        }
```

In diesem Beispiel wird beim Aufruf der Methode zwei Parameter verlangt, zwei Ganzzahlen (Integer), die der Methode übergeben werden. In dem Methodenrumpf (zwischen den geschweiften Klammern) werden diese Zahlen miteinander multipliziert und das Ergebnis über den Befehl *return* an die Methode zurückgeliefert.

3.2.1.3. Konstruktoren

Wie oben erwähnt werden Klassen über Konstruktoren instantiiert. Konstruktoren sind in ihrem Aufbau den Methoden ähnlich. Sie werden bei der Instanzbildung eines Objektes automatisch aufgerufen.
Konstruktoren tragen immer denselben Namen wie die Klasse in der sie sich befinden. Da der Konstruktor nur der Instantiierung einer Klasse dient, hat dieser keine Rückgabewerte, wohl aber Übergabeparameter, die beim Aufrufen des Konstruktors übergeben werden[40].

[39] Vgl. Wolff, Christian: Einführung in Java, S. 118
[40] Vgl. Wolff, Christian: Einführung in Java, S. 120

Eine Instantiierung der Klasse *MeineKlasse* sähe dann wie folgt aus:

```
MeineKlasse ersteKlasse = new MeineKlasse (ÜbergabeParameter);
```

Der erste Ausdruck dieser Befehlszeile *MeineKlasse* ist der Name der Klasse, von der ein Objekt gebildet werden soll.
Der Bezeichner *ersteKlasse* ist der Name, der dem Objekt zugewiesen wird.
Rechts des Gleichheitszeichen ist der Operator *new*, der bei der Instantiierung eingesetzt wird und der Konstruktor, an den mögliche Parameter übergeben werden.

Ein wichtiges Konzept für Konstruktoren und Methoden ist das Überladen von Methoden. In einer Klassen lassen sich mehrere Methoden oder Konstruktoren mit dem selben Namen definieren. Sie müssen aber bei Namensgleichheit zur eindeutigen Identifizierbarkeit unterschiedliche Übergabeparameter aufweisen[41].

In diesem Kapitel wurden die Komponenten, aus denen eine Klasse besteht, kurz vorgestellt.
Der nächste Abschnitt handelt von weiteren wichtigen Konzepte, die in Java verwirklicht sind.

3.2.2. Vererbung

Das Konzept der Vererbung ist ein mächtiges Instrument in der Objektorientierung. In Java kann von einer vorhandenen Klasse eine andere abgeleitet werden. Diese abgeleitete Klasse erbt dann alle Eigenschaften und Methoden der Oberklasse. Durch dieses System lassen sich in Java ganze Klassenhierarchien bilden. Eine Klasse kann aber nur von einer Klasse abgeleitet werden, Mehrfachvererbung ist in Java, mit der Ausnahme der Interfaces (s.a. weiter unten), nicht erlaubt[42].

[41] Vgl. Wolff, Christian: Einführung in Java, S. 118
[42] Vgl. Wolff, Christian: Einführung in Java, S. 123f

Durch die Vererbung lassen sich vorhandene Klassen bequem erweitern, da man die Methoden der Oberklasse übernimmt und nur noch weitere benötigte Methoden programmieren muss.

Der korrekte Befehl zum Ableiten einer Klasse lautet:

```
Public class unterklasse extends oberklasse
```

Die Klasse *unterklasse* erbt also die Methoden und Eigenschaften der Klasse *oberklasse*.

Ausnahmen von der Vererbung bilden Klassen, die mit *final* gekennzeichnet sind. (*final class* ...) Von diesen Klassen kann nicht abgeleitet werden, sie bilden das Ende einer Klassenhierarchie.
Das Gegenteil von *final* ist eine *abstract class*. Eine abstrakte Klasse muss abgeleitet und alle Methoden der abstrakten Klassen müssen in der Unterklasse überschrieben werden, um sie verwenden zu können[43].
Eine weitere Ausnahme bilden Methoden, die mit *private* gekennzeichnet, denn diese Methoden nicht vererbt werden.

Das überschreiben von Methoden bezeichnet man als Polymorphie und stellt ein wichtiges Konzept der objektorientierten Programmierung dar. Geerbte, überschriebene Methoden tragen denselben Namen wie die ursprüngliche Methode der Oberklasse. Eine überschriebene Methode der Oberklasse wird mit dem Schlüsselwort *super* aufgerufen.

3.2.3. Interfaces

Wie oben bereits erwähnt sind Mehrfachvererbungen, d.h. eine Klasse erbt Eigenschaften und Methoden von mehreren Oberklassen, nicht zulässig. Die Ausnahmen dieser Regel bilden die Interfaces.

Ein Interface ist einer Klasse ähnlich aufgebaut. Es wird mittels des Schlüsselwortes *interface* deklariert. Jedoch besteht es ausschließlich aus abstrakten Methodendeklarationen ohne Methodenrumpf und Interfaces können keine Variablen definieren, sondern nur Konstanten[44].

[43] Vgl. Wolff, Christian: Einführung in Java, S. 128

Das Ableiten einer Klasse von einem Interface erfolgt über den Befehl *implements*:

```
public class meineKlasse implements meinInterface { }.
```

Um eine Fehlermeldung bei der Kompilierung einer Klasse zu vermeiden, die von einem Interface abgeleitet ist, muss jede Methode des Interfaces von der Klasse überschrieben werden[45].

3.3. Datentypen

Die Variablen, die in Java vereinbart werden, haben immer einen bestimmten, wohldefinierten Typ. Prinzipiell lassen sich zwei Arten von Datentypen in Java unterscheiden, die primitiven bzw. einfachen Datentypen und die Referenzdatentypen.
Im folgendem Kapitel werden zuerst die einfachen, später die Referenzdatentypen näher betrachtet.

3.3.1. Einfache Datentypen

Jeder einfache Datentyp in Java hat einen vorgegebenen Wertebereich, der von einer Variablen dieses Typs nicht überschritten werden darf.
Bei der Deklaration einer Variable eines bestimmten einfachen Datentyps stellt die interne Speicherverwaltung den benötigten Platz an einer bestimmten Stelle zur Verfügung. Bei der Initialisierung, d.h. bei der Zuweisung eines Wertes an die Variable, wird dieser Wert an die vorher von der Speicherverwaltung festgelegte Stelle geschrieben[46].

---

[44] Vgl. Louis, D., Müller, P. : Jetzt lerne ich Java, S.165
[45] Vgl. Louis, D., Müller, P. : Jetzt lerne ich Java, S.166
[46] Vgl. Louis, D., Müller, P. : Jetzt lerne ich Borland Jbuilder 4, S.67f

In der folgenden Tabelle werden die primitiven Datentypen von Java aufgeführt:

Datentyp	Beschreibung	Speicherplatz	Wertebereich
boolean	Boolescher Wert	1 Bit	true oder false
char	Zeichen, Buchstabe	16 Bit	Unicode-Werte
byte	Ganze Zahl	8 Bit	-128 bis +127
short	Ganze Zahl	16 Bit	-32768 bis +32767
int	Ganze Zahl	32 Bit	-2147483648 bis +2147483647
long	Ganze Zahl	64 Bit	-2^{63} bis $+2^{63}-1$
float	Fließkommazahl	32 Bit	-3,40282347E+38 bis +3,40282347E+38
double	Fließkommazahl	64 Bit	-1,7976931348623157E +308 bis +1,7976931348623157E+308

Tabelle über die einfachen Datentypen[47]

Annmerkungen zu der Tabelle:

Der Boolesche Wert kann nur die Zustände wahr(true) oder falsch (false) annehmen. Er wird als Ergebnistyp bei logischen Vergleichen eingesetzt.

Der Unicode, Wertebereich von char, umfasst 65536 Zeichen. Im ihm sind alle diversen Umlaute der gängigen Sprache und auch japanische und chinesische Schriftzeichen enthalten.[48]

[47] Vgl. Louis, D., Müller, P. : Jetzt lerne ich Java, S. 60 und
Flanagan, D.: Java in a nutshell, S. 25f

[48] Vgl. Louis, D., Müller, P. : Jetzt lerne ich Borland Jbuilder 4, S.71

3.3.2. Operatoren

Ein wichtiges Werkzeug, um mit den oben beschriebenen Datentypen arbeiten zu können, sind die Operatoren. Diese Operatoren erfüllen bestimmte Aufgaben, die in der nachfolgenden Tabelle aufgeführt werden.

Operator	Beschreibung	Beispiel
=	Zuweisung	Weist einer Variablen einen Wert zu: x=5;
+= -= *= /=	Zuweisung mit Operation	x+=5 entspricht x=x+5;
++ --	Inkrement, Dekrement	Erhöht oder erniedrigt eine Variable um eins x++; entspricht x=x+1;
!	logische NICHT	Negiert den Wahrheitsgehalt einer Aussage. Einsatz besonders in Kontrollstrukturen
* /	Multiplikation, Division	x=5*4;
%	Modula-Division	8%3 liefert als Ergebnis 2
- +	Subtraktion, Addition	x=5-2;
< <= >= >	Vergleichsoperatoren	Zum Vergleich zweier Werte, Rückgabewert ist true oder false
= =, !=	Vergleich (gleich,ungleich)	Zum Vergleich auf Gleichheit oder Ungleichheit, liefert true oder false zurück.
&&	logisches UND	Verknüpfung zweier Aussagen, liefert true, wenn beide Aussagen zutreffen: if ((x>5) &&(y<10))
||	logisches ODER	Verknüpfung zweier Aussagen, liefert true, wenn eine Aussage zutrifft: if ((x<15) || (y>4))

Tabelle zu Operatoren[49]

[49] Vgl. Louis, D., Müller, P. : Jetzt lerne ich Java, S. 65f und
Flanagan, D.: Java in a nutshell, S. 36f
Diese Tabelle ist bei weitem nicht vollständig, es wurden nur die wichtigsten Operatoren aufgeführt

3.3.3. Referenz-Datentypen

Die komplexen Datentypen in Java sind Objekte und Arrays. Diese Datentypen werden Referenzdatentypen genannt, weil sie „per Referenz“ („by references“) verarbeitet werden, d.h. wird ein Objekt mit einem bestimmten Namen gebildet, so wird an die Speicherstelle des Namens eine Referenz auf die eigentliche Speicherstelle des Objektes vergeben. Wird nun einer Methode ein Objekt übergeben, so wird nicht das gesamte Objekt übergeben, sondern nur seine Referenz.[50]
Die Referenzdatentypen in Java umfassen Arrays, Klassen und Interfaces. Klassen und Interfaces wurden bereits im vorangegangenem Kapitel erläutert, deshalb wird hier ein besonderes Augenmerk auf die Arrays gerichtet.

Ein Array kann sowohl aus primitiven Datentypen, als auch aus Objekten gebildet werden.
Als Beispiel wird ein Array aus Integer-Variablen gebildet.

```
int[] feld;
feld = new int[10];
```

Im ersten Schritt wird der Array unter dem Namen „feld“ deklariert. Im zweiten Schritt wird die Menge der Elemente, die dieser Array haben soll (oben 10), festgelegt. Diese zwei Schritte lassen sich zu einem Schritt zusammenfassen :

```
int[] feld = new int[10];
```

Nun kann der Array initialisiert werden, seine Elemente können mit Zahlen gefüllt werden:

```
feld[] = {1,2,3,4,5,6,7,8,9,10};
```

Bei dem Zugriff auf die einzelnen Elemente des Arrays ist zu beachten, dass das erste Element *feld[0]* ist (im Beispiel also die 1).

[50] Vgl. Flanagan, D.: Java in a nutshell, S. 27f

Des weiteren lässt sich ein Array nicht nur aus primitiven Datentypen, sondern auch aus Instanzen bilden. Die formale Erstellung erfolgt analog wie oben gesehen.

Schließlich können mehrdimensionale Arrays erstellt werden. Hier das Beispiel eines zweidimensionalen Array, wiederum aus Integervariablen[51]:

```
int[][] matrix = new int[2][4];
```

Die Initialisierung erfolgt in diesem Fall über die paarweise Eingabe der Zahlen:

```
matrix[][] = { {1,2}, {2,6},{234,46},{4533,456} };
```

3.4. Kontrollstrukturen

In diesem das Kapitel 3 abschließenden Unterpunkt wird auf die Möglichkeiten hingewiesen , den Programmfluss einer Java Applikation durch Bedingungen und Schleifen zu gestalten.

3.4.1. Bedingungen

Bedingungen sind ein sehr wichtiges Instrument zur Steuerung des Programmflusses in einem Javaprogramm. Der Zweck einer Bedingung ist es, aufgrund des Wahrheitsgehaltes dieser Bedingung, einen Pfad im Quellcode zu wählen. Prinzipiell werden zwei Arten von Bedingungen unterschieden, die *if*-Bedingung und die *switch*-Bedingung.

[51] Vgl. Wolff, Christian: Einführung in Java, S. 60

Die if - else - Bedingung

Die *if-else*-Bedingung ist eine einfache Verzweigung im Programm. Die *if*-Anweisung fragt die Bedingung ab. Ist diese erfüllt, werden die Anweisung des *if*-Blocks ausgeführt, ist diese Bedingung nicht erfüllt, so werden die Anweisungen des *else*-Blocks ausgeführt. Die *else*-Verzweigung ist aber optional, also nicht zwingend notwendig. Ist eine *if*-Anweisung, ohne *else*-Block, nicht erfüllt, so wird im Programm der *if*-Block übersprungen und der nächste Befehl ausführt.[52]

Die Syntax dieser Bedingung lautet[53]:

```
if (Bedingung)
    {
     //Bedingung ist wahr
     Anweisungen;
     }
  else
    {
     //Bedingung ist falsch
     Anweisungen;
     }
```

Aus diesem Beispiel wird klar, das in der *if*-Bedingung nur solche Ausdrücke abgefragt werden können, die einen Wahrheitswert zurückliefern. Das wären sowohl boolesche als auch logische Ausdrücke. Ein Beispiel einer *if*-Bedingung mit einer logischen Bedingung.

```
int a,b,;

if (a>=b)

{   System.out.println ("a ist größer oder gleich b");    }

else

{  System.out.println ("a ist kleiner b");     }
```

[52] Vgl. Wolff, Christian: Einführung in Java, S. 81

[53] Vgl. Louis, D., Müller, P. : Jetzt lerne ich Java, S. 65

Die switch-Bedingung

Mit *if-else*-Anweisung lässt sich ein Programm nur in zwei Richtungen gabeln. Können nun mehr als zwei mögliche Fälle unterschieden werden, eignet sich eine *if-else*-Anweisung nicht, da jeder mögliche Fall mit einer eigenen *if*-Bedingung abzufragen wäre. Aus diesem Grunde gibt es in Java die sogenannte *switch*-Bedingung. Das folgende Beispiel verdeutlicht die Syntax dieser Anweisung:

```
char option;
switch (option)
{
case 'D': System.out.println ("Datei öffnen");
break;
case 'B': System.out.println („Dokument bearbeiten");
break;
case 'A': System.out.println („Ansicht öffnen");
break;
default: System.out.println („falsche Option gewählt);
break;
}
```

Erläuterungen: Zu Beginn wird eine Variable option vom Typ *char* vereinbart. Diese Variable wird dann über die *switch*-Bedingung von jedem Fall (*case*) abgefragt. Trifft sie für einen bestimmten Fall zu, so wird der Anweisungsblock dieses Falles ausgeführt. In dem Beispiel oben wäre die Ausführung jeweils eine Textausgabe.
Die break-Anweisung am Ende jedes Anweisungsblocks ist eine Abbruchbedingung, d.h. erhält das Programm den *break*-Befehl, so springt es zum Ende der *switch*-Anweisung.
Wäre in unserem Beispiel keine *break*-Befehle eingebaut und die *switch*-Bedingung würde mit dem ersten Fall (case 'D') übereinstimmen, so würden auch alle Anweisungen der nachfolgenden Fälle ausgeführt[54].

[54] Vgl. Wolff, Christian: Einführung in Java, S. 82

Sollte kein Fall die *switch*-Bedingung erfüllen, so wird optional der default-Befehl aufgerufen.
Sie *switch*-Bedingung ist immer eine Konstante und kann nur Ganzzahlwerte annehmen (*char, byte, short, int*), keine Gleitkommazahlen, *long*-Variablen oder *boolesche* Datentypen.

3.4.2. Schleifen

Häufig kommt es in einem Programm vor, das bestimmte Anweisungen mehrmals hintereinander ausgeführt werden müssen. Um nicht jede Ausführung einzeln im Programm ausschreiben zu müssen, gibt es in Java Schleifen. Diese Schleifen lassen sich in drei Kategorien unterteilen: while-, do/while- und for-Schleifen[55].

While-Schleifen

While-Schleifen prüfen die Bedingung, die zum Durchlauf der Schleife notwendig ist, am Anfang, im Kopf der Schleife. Ist die Bedingung erfüllt, durchläuft das Programm den Schleifenrumpf mit den einzelnen Anweisungen.

Ein Beispiel:

```
int i;
i =0;
while (i<10)
   {
     system.out.println (i);
     i++;
   }
```

Anfangs wird eine Integervariable i deklariert und ihr den Wert 0 zugewiesen. Im ersten Schritt wird im Schleifenkopf geprüft, ob die Bedingung erfüllt ist. Da die Bedingung erfüllt ist (null ist kleiner zehn), wird nun der Schleifenrumpf durchlaufen. Die Anweisungen im Schleifenrumpf geben zuerst den derzeitigen Wert der Variablen aus und anschließend wird

[55] Vgl. Wolff, Christian: Einführung in Java, S. 83f

die Variable i um eins erhöht. Nach der Bearbeitung des Schleifenrumpfes springt das Programm wieder an den Schleifenkopf und prüft erneut die Bedingung.

Dies wird solange ausgeführt, bis die Bedingung nicht mehr erfüllt ist. (Im Fall oben wäre der Abbruch der Schleife bei i=10 gegeben.)

Bei solchen Schleifen ist zu beachten, dass die Iterationsschritte (Erhöhen oder erniedrigen von Variablen) und die Abbruchbedingung so gewählt sind, dass keine Endlosschleifen entstehen.

Do/while-Schleifen

Diese Art von Schleifen ähneln den while-Schleifen. Der Unterschied besteht darin, dass do/while-Schleifen die Schleife mindestens einmal durchlaufen, bevor die Abbruchbedingung geprüft wird[56].

Ein Beispiel:

```
int i=10;
do
{
system.out.println(i);
 i+=2;
}
while (i<10);
```

Dieses Beispiel liefert als Ergebnis die Ausgabe der Zahl 10. Die Schleife wird hier genau einmal durchlaufen, da die Abbruchbedingung schon erfüllt ist.

[56] Vgl. Wolff, Christian: Einführung in Java, S.85

For-Schleife

Die oben angeführte Beispielschleifen sind Zählschleifen. Diese Schleifen können in Java auch durch for-Schleifen dargestellt werden. *For*-Schleifen haben in ihrem Schleifenkopf alle Informationen, die nötig sind werden festgelegt , sowohl eine deklarierte Variable kann dort initialisiert werden, als auch die Abbruchbedingung und der Iterationsschritt, mit dem die Variable geändert werden soll. Diese Schleife ist wie die while-Schleife, kopfgesteuert, d.h. die Abbruchbedingung wird vor dem ersten Durchlauf geprüft[57].

Ein Beispiel

```
int i;
for (i=0; i<10; i++)
{
system.out.println (i);
}
```

Das Ergebnis ist dasselbe wie bei dem Beispiel zur *while*-Schleife.

Die Anwendung der einzelnen Schleifen ist von Präferenzen des Programmierers abhängig, so lassen sich für die selben Probleme sowohl mit der while, als auch mit der for-Schleife lösen.

[57] Vgl. Wolff, Christian: Einführung in Java, S.85f

4. JDBC

4.1. Einleitung

Um über Java auf Datenbanken zugreifen zu können, wurde ab dem JDK 1.1.4. (Java Development Kit) ein von Sun Microsystems entwickeltes Paket ausgeliefert, die JDBC (Java Database Connectivity).

JDBC ist ein spezifiziertes Verfahren, das in Java geschrieben eine einheitliche Schnittstelle zu Datenbanken darstellt[58]. Die Funktionsweise ist der von ODBC (Open Database Connectivity) ähnlich.

ODBC ist eine standardisierte Schnittstelle zum Zugriff auf Datenbanken mit Hilfe von SQL-Statements. Es ist eine weitverbreitete Schnittstelle, die durch das X/Open-Gremium als SQL CLI (Call Library Interface) standardisiert ist. Der Hauptunterschied zwischen ODBC und JDBC besteht darin, dass sich ODBC an den Programmiersprachen C und C++ orientiert, während JDBC komplett in Java geschrieben. Da JDBC auch die Java-Ausnahmen verwendet ist es robuster als ODBC. Eine Java-Applikation, die mit JDBC-Treibern arbeiten kann, hat ebenfalls Performancevorteile.

JDBC stellt wie ODBC nur den sogenannten Treibermanager zur Verfügung, der das Grundgerüst für die Treiber darstellt. Die Treiber werden von Datenbankherstellern oder von Middleware-Spezialisten zur Verfügung gestellt. Deshalb existiert nicht für jedes Datenbanksystem ein nativer Treiber in Java. Aus diesem Grunde gibt es im JDK eine JDBC/ODBC-Bridge, die Datenbanken über eine Java-Applikation anbinden kann, ohne dass ein in Java geschriebener (native) Treiber existiert. Weiterführende Erklärungen zum Treiber-konzept in JDBC kommen später in diesem Kapitel. Des weiteren werden in diesem Kapitel die Architektur von JDBC-Anwendungen und die einzelnen Schritte für den Aufbau eines JDBC-Programmes erläutert.

[58] Vgl. Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.107

4.2. Die Architektur einer JDBC-Anwendung

Der Zugriff auf eine Datenbank in einem Netzwerk lässt sich über two-tier- und über three-tier-Architekturen verwirklichen. Diese beiden Konzepte werden nun vorgestellt.

4.2.1. Two-Tier-Architektur

Ausgehend von der strengen Sicherheitsmaßnahme für Applets[59], bzw. Applikation, die nur Verbindung mit dem Rechner in Verbindung treten kann, von dem es geladen wurde, entstehen two-tier-Architekturen, die im wesentlichen dem klassischen Client-Server-Aufbau entspricht.
Es gibt zwei Realisierungformen dieser Architektur.

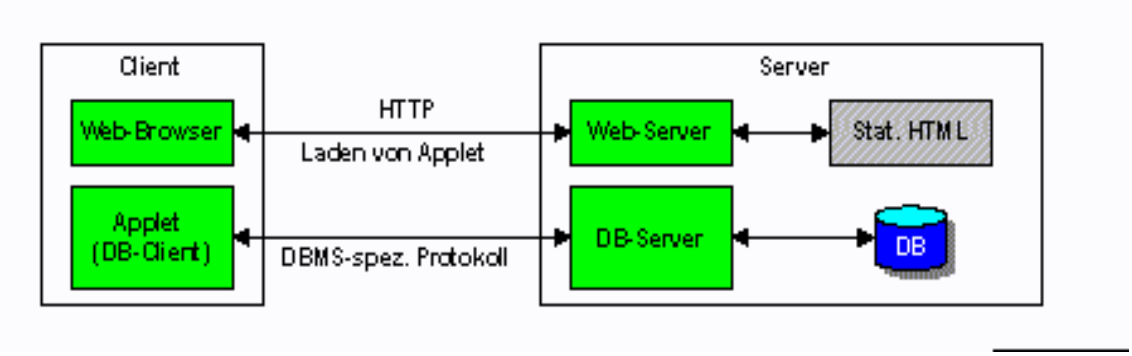


Abb.: Two-Tier-Architektur mit nativem Java-DB-Client (vgl."Buisness online",10/97)[60]

Dies ist ein clientseitiger Zugriff auf die Datenbank (DB). Im ersten Schritt lädt sich der Client vom Server ein Java-Applet, z.B. via Internet herunter. Dieses Applet verbindet den Client mit der Datenbank über ein DBMS (Datenbankmanagementsystem)-spezifisches Protokoll (z.B. JDBC)[61].

[59] Applets sind Java-Programme, die nicht unabhängig lauffähig sind, sondern nur eingebettet, z.B. in einer HTML-Webseite, funktionsfähig sind.
[60] Vgl Hartwig, J.: Einführung in JDBC, Kap.3
[61] Vgl Hartwig, J.: Einführung in JDBC, Kap.3

Die zweite Möglichkeit einer Two-Tier-Architektur:

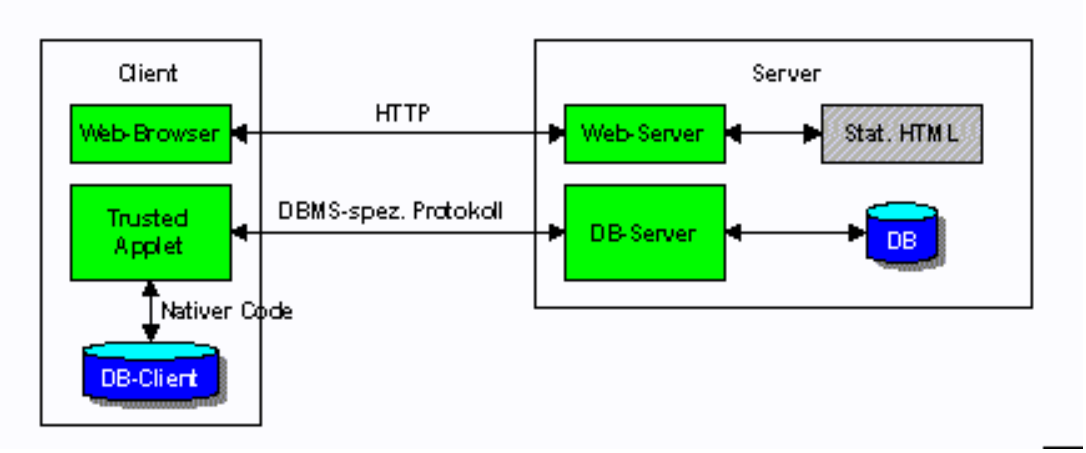


Abb.: Two-Tier-Architektur mit nativem DB-Client (vgl."Buisness online",10/97)[62]

Dieses Modell sieht dem oben ähnlich, aber der Client lädt sich hier ein „Trusted Applet" vom Server. Dieses Applet stellt über einen nativen DBMS-Treiber (z.B. JDBC-ODBC-Bridge) eine Verbindung zur Datenbank her.

Eine Two-Tier-Architektur setzt auf der Serverseite einen leistungsstarken Rechner voraus, der sämtliche Operationen ausführen kann, wie z.B. Datenverbindung aufbauen, nach dem Aufbau der Verbindung Anfragen entgenehmen und verarbeiten und die Ergebnisse zurückliefern. Ist ein leistungsstarker Server nicht vorhanden kann eine Three-Tier-Architektur angewandt werden, die diese Aufgaben auf zwei Server verteilt.

4.2.2.Three-Tier-Architektur

Bei dieser Konstruktion findet eine Trennung des Servers mit der Datenbankanbindung und dem eigentlichen Datenbankserver statt. Folgende Grafik verdeutlicht dies:

[62] Vgl. Hartwig, J.: Einführung in JDBC, Kap.3

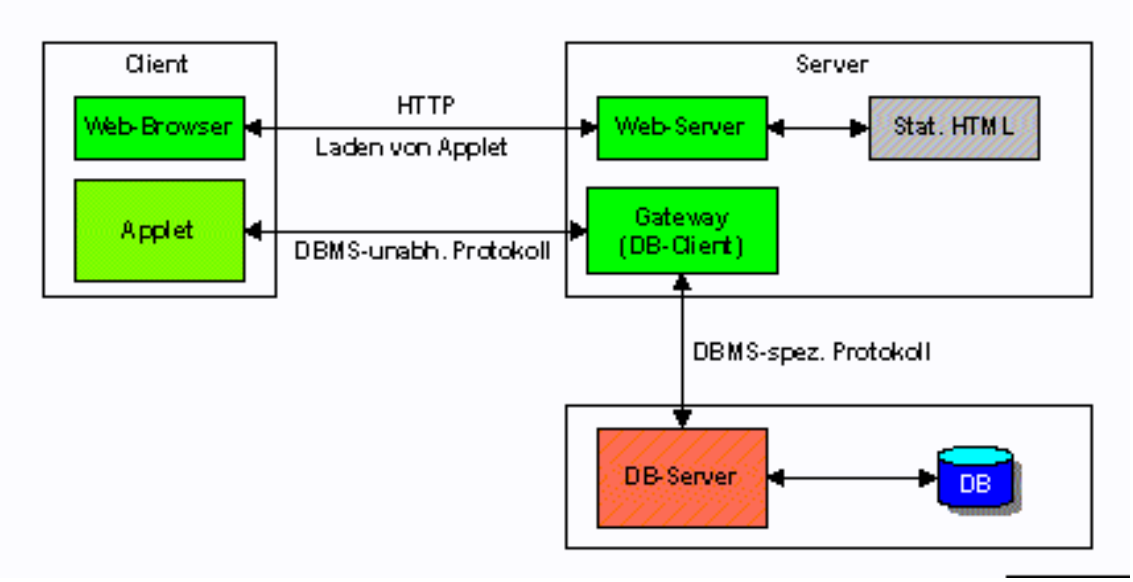


Abb.: Three-Tier-Architektur (vgl. "Business Online", 10/97)[63]

Auch hier wird vom Server ein Applet heruntergeladen. Dieses Applet verbindet dann den Client mit dem Gateway, dieser wiederum verbindet sich mit einem DBMS-spezifischen Protokoll des Datenbankservers.
Der Vorteil dieser Architektur liegt in der Entlastung des Servers durch die Trennung von Datenbank- und Webserver. Auch wird eine Verbesserung der Datenbanksicherheit erreicht, aufgrund der Tatsache, dass der Client nicht mehr direkt auf die Datenbank zugreift, sondern über ein Gateway.

Das war ein Einblick in die Möglichkeiten, eine Datenbankanbindung mit Java über ein Netzwerk zu gestalten. Im kommenden Kapitel werden die verschiedenen Treibertypen von JDBC erklärt.

63 Vgl. Hartwig, J.: Einführung in JDBC, Kap.3

4.3. Das JDBC-Treiberkonzept

Ein Treiber ist eine Softwarekomponente, die die Kommunikation zwischen einer Datenbankanwendung und der entsprechenden Zieldatenbank ermöglichen soll. Grundsätzlich muss ein JDBC-Treiber die JDBC-Funktionalität und den SQL-Sprachumfang unterstützen.
Es werden vier verschiedene Arten von JDBC-Treibern unterschieden.

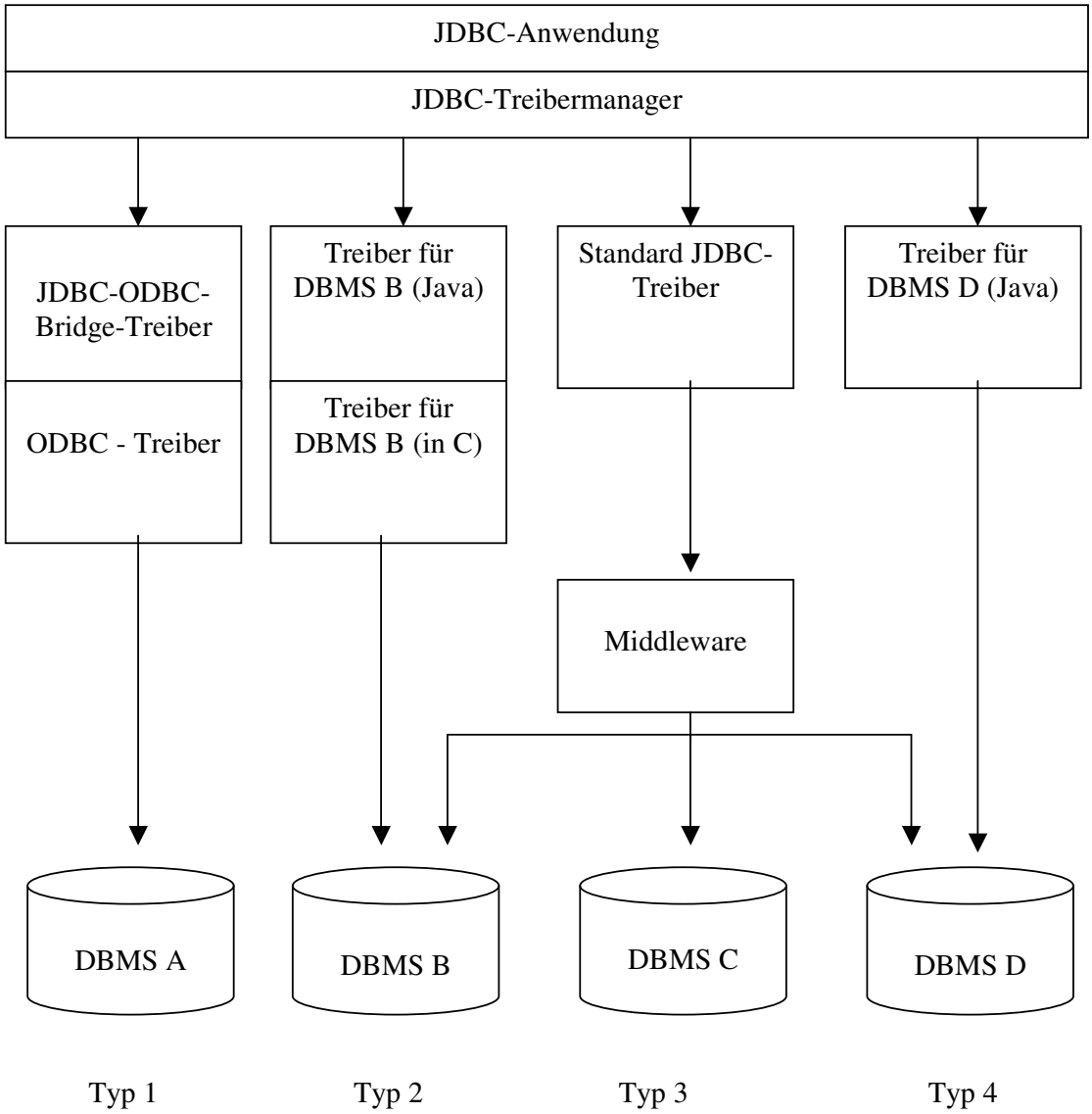


Abb.: Die vier JDBC-Treibertypen[64]

[64] Abbildung nach Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.111

Typ 1-Treiber

Bei dieser Art der Treiber wird einem ODBC-Treiber, der gewöhnlich in der Programmiersprache C geschrieben ist, ein JDBC-Treiber aufgesetzt. Diese Methode ist recht einfach und es lassen sich sehr viele Datenbanken anbinden, da ODBC-Treiber weit verbreitet sind. Ein gravierender Nachteil ist, dass die Anbindung einer solchen Datenbank via Internet nicht praktikabel ist, weil der ODBC-Treiber vom Client installiert werden muss. Zudem ist der ODBC-Treiber meist in C geschrieben, wodurch er nicht portabel ist[65].

Micorsoft Access-Datenbanken müssen auf diese Weise an eine Java-Applikation angebunden werden.

Typ 2-Treiber

Dieser Typ ist dem Typ 1-Treiber sehr ähnlich, nur anstelle eines ODBC-Treibers wird ein herstellerspezifischer Treiber verwendet. Auf diesen, nicht in Java programmierten, Treiber wird eine Java-Schicht aufgesetzt. Diese ruft Funktionen einer Zugriffs-schnittstelle auf, mit der ein Datenbankzugriff ermöglicht wird. Das ist für den Hersteller eine bequeme und praktische Methode, seine Datenbanken javafähig zu machen. Diese Treiber sind aber herstellerabhängig, d.h. mit diesen Treibern können nur die Datenbanken des Herstellers angebunden werden.
Dieser Treiber ist nicht internetfähig, weil der herstellerspezifische Teil des Treibers in C geschrieben ist und somit nicht portabel ist[66].

Als Beispiel sei Oracle erwähnt, die einen solchen Treiber zur Anbindung der Oracle-Datenbanken über ein Javaprogramm verwenden.

[65] Vgl . Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.110
[66] Vgl . Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.110

Typ 3-Treiber

Dieser Treiber ist ein komplett in Java geschriebener Netztreiber, der eine Verbindung zur Middleware herstellt, die dann alle weiteren Aufgaben übernimmt. Der Treiber muss nicht auf dem Client-Rechner installiert werden, sondern wird von der Middleware (auf Serverseite) geladen. Damit ist er der flexibelste der vier Treibertypen.
Die Middleware muss nicht notwendigerweise in Java geschrieben sein, sondern es ist nur ein von Java und der Middleware unterstüztes Kommunikationsprotokoll vonnöten[67].
Diese Konstruktion entspricht der Three-Tier-Architektur, die weiter oben erklärt wurde.

Typ 4-Treiber

Die komplett in Java programmierten Typ 4-Treiber gelten als die modernsten. Sie entsprechen dem ersten Two-Tier-Architekturmodell, das bereits beschrieben wurde. Das strenge Java-Sicherheitskonzept verlangt, dass die Datenbank auf demselben Rechner gespeichert ist wie der Webserver.
Die reinen Java-Treiber sollen mit der Zeit die Typ 2-Treiber ablösen, die nur als Zwischenlösung gedacht sind.
Der Vorteil dieser Treiber liegt, neben der 100%-igen Kompalität mit Java, im problemlosen Laden dieser über das Internet, denn es wird keine weitere Software auf Clientseite benötigt[68].

[67] Vgl . Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.110f

[68] Vgl . Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.112

4.4. Der Aufbau eines JDBC-Programmes

In diesem Unterpunkt wird schrittweise ein JDBC-Programm erstellt. Die folgende Grafik soll einen ersten Überblick über diese Schritte geben.

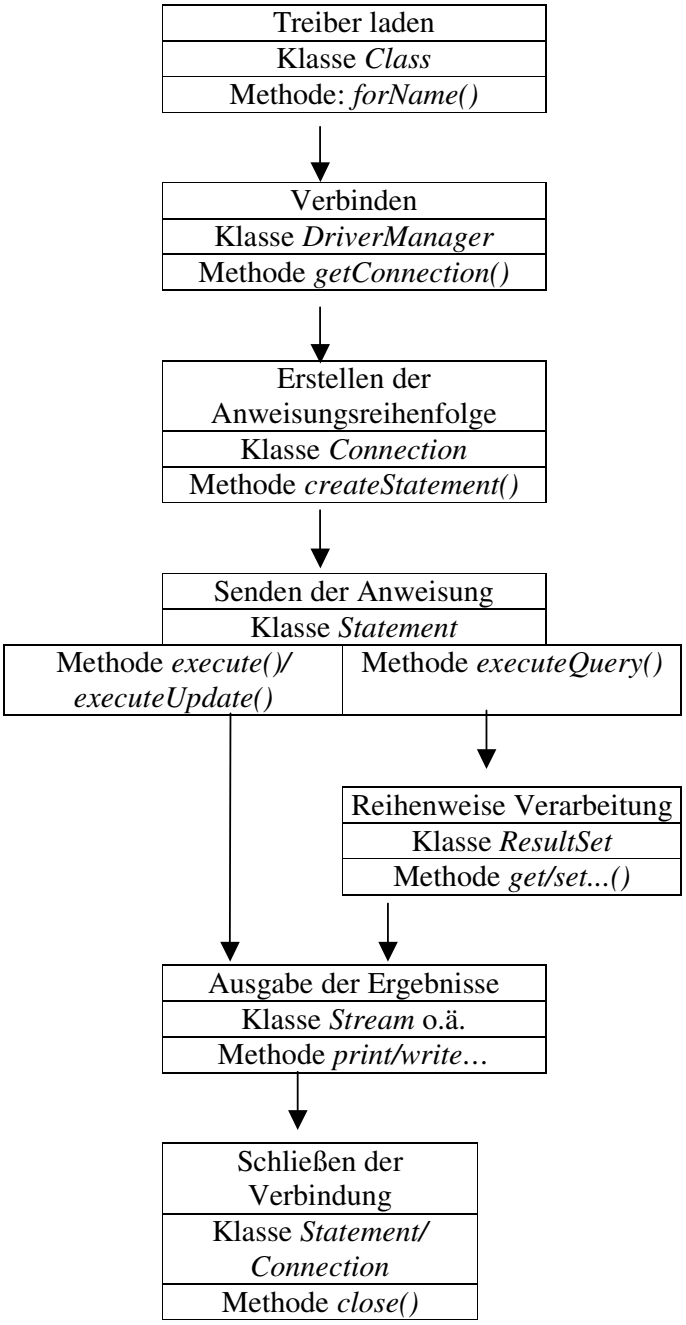


Tabelle nach Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.114

4.4.1. Das Laden des Treibers

Der erste Schritt zur Herstellung einer Verbindung zu einer Datenbank ist das Laden eines Treibers. Anschließend wird der Treiber initialisiert und mit Hilfe des Treibermanagers eine Verbindungsinstanz gebildet, die während der gesamten Verbindung aktiv bleibt.

Das Laden des Treibers kann auf zwei mögliche Arten erfolgen. Entweder ein Treiber wird explizit oder automatisch geladen. Zuerst wird das explizite Laden eines Treibers erläutert[69].

Explizites Laden

Das explizite Laden kann auf zwei Arten erfolgen und zwar durch den Aufruf der Methode *forName* der Klasse *Class* oder durch die Instantiierung der Klasse *Driver*. Beide Möglichkeiten werden hier aufgeführt.

Die Klasse *Class* ist eine statische Klasse, deren Syntax wie folgt aussieht:

public static native Class forName (String forName)
throws ClassNotFoundException; [70]

Der Befehl throws *ClassNotFoundException* stellt eine Ausnahmebehandlung dar, die eintritt, wenn eine Klasse nicht gefunden wurde.
Um einen Treiber mit dieser Methode zu laden, muss der vollständige qualifizierte Name an die Methode *forName* übergeben werden. Diese lädt den benannten Treiber dann in den Java-Interpreter[71].
Das Laden eines Treiber, hier eines ODBC/JDBC-Treibers (Typ 1-Treiber), entspricht dem folgendem Beispiel:

Class.forName(“sun.jdbc.odbc.jdbcodbcDriver“);

[69] Vgl . Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.113
[70] Vgl. Flanagan, D.: Java in a nutshell, S.467
[71] Vgl. Flanagan, D.: Java in a nutshell, S.467

Die zweite Methode, eine Treiber explizit zu laden, verwendet eine Instanz der Klasse *Driver* mit der in dieser Klasse verwirklichten Methode *registerDriver()*. Hier ein kleines Beispiel dieser Möglichkeit:

```
Driver drv = new COM.ibm.db2.jdbc.net.DB2Driver();
DriverManager.registerDriver(drv);
```

Hier wird ein nativer Javatreiber (Typ 4-Treiber) für ein DB2-Datenbanksystem geladen. Mit dem zweiten Befehl wird der Datenbankmanager über die Klasse DriverManager beim System registriert.
Der Vorteil ist die Bekanntgabe einer Fehlermeldung schon zur Übersetzungszeit des Programms, sollte der angegebene Treiber nicht gefunden werden[72].

Automatisches Laden

Das automatische Laden von Treibern wird von JDBC unterstützt. Dabei wird in der jdbc.drivers Systemeigenschaft, der Property, festgelegt, welche Treiber beim Start automatisch geladen werden.

4.4.2. Verbindung zur Datenbank herstellen

Die wichtigste Klasse bei einer Datenbankverbindung ist die Klasse *DriverManager*, sie stellt den Grunddienst bei der Verwaltung der JDBC-Treiber. Die *DriverManager*-Klasse implementiert den JDBC-Treibermanager, der wiederum einen oder mehrere JDBC-Treiber verwaltet. Der *DriverManager* ist also eine Verwaltungschnittstelle zwischen der Anwendung und den Treibern.

[72] Vgl . Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.115

Die einzige Information, die bei einer Java-Datenbankapplikation von JDBC verlangt wird, ist die URL (Uniform Ressource Locator) der Datenbank, damit diese eindeutig von JDBC gefunden wird.

Die eigentliche Verbindung von der Anwendung wird über eine Instanz der *Connection*-Schnittstelle hergestellt. Diese Schnittstelle (Interface) erstellt mit Hilfe der URL, Kennung und Kennwort (beide optional) eine Verbindung zur Datenbank.
Die *Connection*-Sitzung umfasst die SQL-Anweisungen, die an die Datenbank gesendet werden und die Rückgabe der Ergebnisse der SQL-Anweisungen.

Nun aber zurück zur Klasse *DriverManager*, die unter anderen die Methode *getConnection()* enthält, welche die Verbindung zur Datenquelle erstellt, die durch die URL gekennzeichnet ist. Beim Aufruf dieser Methode werden alle registrierten Treiber darauf getestet, ob sie eine Verbindung zur Datenbank erstellen können. Der erste Treiber, den die URL erkennt, baut die Verbindung auf.

Die Methode *getConnection* hat drei Ausprägungen:

```
getConnection (String URL);
getConnection (String URL, String kennung, String kennwort);
getConnection (String URL, Properties info);
```

Die erste Variante benötigt nur einen String (Zeichenkette) mit der URL. Optional können in der zweiten Variante eine Kennung und ein Kennwort für eine Datenbank mit eingebaut werden. Die dritte Variante spielt bei diesen Betrachtungen keine Rolle[73].

[73] Vgl . Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.117f

Um die Funktionsweise eines Verbindungsaufbaues zu verdeutlichen, wird nun ein Beispiel angeführt.

```
String URL = “jdbc:odbc:mitarbeiter”;
Connection verbindung = null;
verbindung = DriverManager.getConnection (URL);
```

Im ersten Schritt wird ein String mit der URL einer Datenbank namens 'mitarbeiter' definiert. Die Standardsyntax der URL lautet[74]:

```
jdbc:<subprotocoll>:<datenbankname>;
```

Anschließend wird eine Instanz ‚verbindung’ der Schnittstelle *Connection* gebildet und initialisiert (auf null gesetzt).
Schließlich wird dieser Instanz die Verbindung zur Datenbank übergeben, die über die *getConnection*-Methode hergestellt wird.
Damit wäre eine Verbindung zu einer Datenbank hergestellt.

4.4.3. SQL-Statement erzeugen und an den Datenbankserver senden

Eine Abfrage und Datenmodifizierungen auf einer Datenbank erfolgen durch Anweisungsobjekte. Ein Anweisungsobjekt ist eine SQL-Anweisungsfolge, die sich in einer Transaktion befindet.
Solche Anweisungsobjekte werden von einer *Statement*-Schnittstelle oder einer ihrer Subkomponenten, implementiert und von Methoden der *Connection*-Schnittstelle erzeugt. Somit ist die *Statement*-Schnittstelle die Basisschnittstelle für alle Klassen, die für die Erstellung von SQL-Statements verwendet werden.

[74] Vgl . Wolff, C.: Einführung in Java, S.369

Das Erstellen einer SQL-Anweisung lässt sich über drei verschiedene Arten ausführen.

Ein einfaches SQL-Statement, das zur Übersetzungszeit des Programmes schon bekannt ist wird mit der *createStatement()*-Methode der *Connection*-Schnittstelle verarbeitet. Ein Beispiel, das auf dem vorhergehenden aufbaut:

```
String URL = "jdbc:odbc:mitarbeiter";
Statement anweisung = null;
Connection verbindung = null;
verbindung = DriverManager.getConnection (URL);
anweisung = verbindung.createStatement();
```

Annmerkungen: Das Beispiel ist eine Erweiterung des Beispiels aus Kapitel 4.4.2.. Zwei neue Anweisungen treten nun auf. Zuerst wird eine Instanz mit dem Namen ‚anweisung' der Schnittstelle *Statement* gebildet. Diese Instanz bereitet über die Zuweisung der Methode *creatStatement()* der Instanz ‚verbindung' ein SQL-Statement vor, welche aus der Schnittstelle Connection gebildet wurde.

Wenn ein SQL-Statement verarbeitet werden soll, das zur Übersetzungszeit des Programmes noch nicht bekannt ist, so wird die *PreparedStatement*-Schnittstelle verwendet , die eine Unterklasse der *Statement*-Schnittstelle ist. Die *PreparedStatement*-Schnittstelle benutzt die *preparedStatement*-Methode der Schnitttstelle *Connection*, um SQL-Statements vorzubereiten, die dann eine effizientere Abarbeitung der Abfragen ermöglicht.

Es gibt in SQL die Möglichkeit, stored procedures[75] zu programmieren. Das sind gespeicherte Prozeduren, die eine Mehrzahl von einzelnen Datenbankoperationen enthalten. Um eine solche Prozedur zu verarbeiten, wird die Klasse *CallableStatement* gebraucht, die eine direkte Unterklasse der *PreparedStatement*-Schnittstelle ist. Diese *CallableStatement*-Schnittstelle verwendet die *prepareCall()*-Methode der *Connection*-Schnittstelle[76].

[75] Zu stored procedures siehe: Ebner, M.: SQL lernen, Kapitel 6
[76] Vgl . Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.119

Mit diesen drei Möglichkeiten lassen sich SQL-Statements erzeugen. Nun sollen diesen Statements ausgeführt werden. Dafür besitzt die *Statement*-Schnittstelle drei weitere Methoden: *execute(), executeQuery(), executeUpdate().* Im folgendem werden diese Methoden im einzelnen erklärt[77].

execute()-Methode

Die *execute()*-Methode liefert nach der Ausführung einer SQL-Anweisung einen booleschen Wert zurück, d.h. diese Methode testet, ob es sich bei der Anweisung um eine Abfrage (*true*) oder um eine andere SQL-Anweisung (*false*) handelt. Diese Methode kann eine SQL-Anweisung prüfen, die keine spezifische Ausgabe hat.

executeQuery()-Methode

Diese Methode führt eine Abfrage auf einer Datenbank aus, d.h. die *executeQuery()*-Methode erwartet als Parameter eine korrekte Select-Anweisung. Ist dies der Fall, so gibt die Methode als Ergebnis eine Instanz vom Typ *ResultSet* zurück. Auf das Thema *ResultSet* wird im nächsten Abschnitt eingegangen.

Als Beispiel für die Anwendung einer solchen Methode wird das Beispiel von oben erweitert.

```
String URL = “jdbc:odbc:mitarbeiter”;
Statement anweisung = null;
Connection verbindung = null;
verbindung = DriverManager.getConnection (URL);
anweisung = verbindung.createStatement();
String statement = “SELECT name, vorname FROM mitarbeiter“;
ResultSet ergebnismenge;
ergebnismenge = anweisung.executeQuery(statement);
```

[77] Vgl . Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.120ff

Anmerkungen: Zuerst wird ein benannter String definiert, der eine für die Datenbank ‚mitarbeiter' korrekte SELECT-Anweisung enthält (die Attribute ‚name' und ‚vorname' der Tabelle ‚mitarbeiter' werden abgefragt),.
Dann wird eine Instanz der Schnittstelle *ResultSet* vereinbart (‚ergebnismenge').
Zum Schluss wird der *executeQuery()*-Methode der String mit dem SQL-Statement als Parameter übergeben. Das Ergebnis liegt, wie oben beschrieben, als Instanz vom Typ *ResultSet* vor und kann deshalb auch nur einer solchen Instanz übergeben werden, in diesem Beispiel der Instanz ergebnismenge.

executeUpdate()-Methode

Das Einsatzgebiet der *executeQuery()*-Methode ist das Einfügen (INSERT), Ändern (UPDATE) und löschen (DELETE) von Datensätzen in einer Datenbank. Als Ergebnis liefert diese Methode die Anzahl der Datensätze, die geändert, eingefügt, bzw. gelöscht wurden.
Ein Beispiel:

```
String URL = "jdbc:odbc:mitarbeiter";
Statement anweisung = null;
Connection verbindung = null;
verbindung = DriverManager.getConnection (URL);
anweisung = verbindung.createStatement();
int anzahl;
anzahl=anweisung.executeUpdate("DELETE FROM mitarbeiter
                                WHERE name= 'Müller');
```

In diesem Beispiel wird auf die bekannte Weise eine Datenbankverbindung aufgebaut und ein SQL-Statement vorbereitet. Anschließend wird eine Ganzzahlvariable (Integer) vereinbart, der in der folgenden Zeile das Ergebnis der Löschaktion übergeben wird, nämlich die Anzahl der gelöschten Datensätze mit dem Namen ‚Müller' in der Tabelle ‚mitarbeiter'.

4.4.4. Verarbeitung der Ergebnisse

Das Ergebnis einer Abfrage wird also, einer Instanz der *ResultSet*-Schnittstelle übergeben. Diese Schnittstelle enthält die Methode *next()* für die Verarbeitung der Ergebnisse. Mit dieser Methode wird die Ergebnismenge, mit Hilfe eines Cursors, reihenweise abgearbeitet, d.h. die Methode *next()* verweist immer auf die nächste Reihe in der Ergebnismenge. Als Rückgabewert liefert sie einen Wert vom Typ boolean. Der Cursor zeigt in der Ergebnismenge auf einen bestimmten Datensatz und es wird überprüft, bei Aufruf der Methode *next(),* ob noch ein Datensatz folgt und liefert *true* zurück, wenn noch ein Datensatz vorhanden und *false*, wenn kein Datensatz mehr vorhanden ist[78].

Das Beispiel von oben wird entsprechend erweitert:

```
String URL = "jdbc:odbc:mitarbeiter";
Statement anweisung = null;
Connection verbindung = null;
verbindung = DriverManager.getConnection (URL);
anweisung = verbindung.createStatement();
String statement = "SELECT name, vorname FROM mitarbeiter";
ResultSet ergebnismenge;
ergebnismenge = anweisung.executeQuery(statement);
while (ergebnismenge.next() )  {
    // Zugriff auf die einzelnen Reihen der Ergebnismenge
}
```

Die einzige Erweiterung ist in diesem Fall eine while-Schleife, die als Bedingung die Methode *next(),* der Instanz ‚ergebnismenge', hat. Die Schleife wird solange durchlaufen, bis *next()* den Wert true zurückliefert, die Abbruchbedingung der Schleife wäre also das Zurückliefern des Wertes *false*, so dass kein Datensatz mehr in der Ergebnismenge ist.
Wie der Zugriff auf die einzelnen Reihen (Datensätze) der Ergebnismenge organisiert ist, wird im nächsten Abschnitt erläutert.

[78] Vgl . Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.122/123

4.4.5. Ausgabe der Ergebnisse

Die Ausgabe der Ergebnisse erfolgt ebenfalls über Methoden der *ResultSet*-Schnittstelle. Der generische Name dieser Methoden, die die einzelnen Spaltenwerte der aktuellen Reihe auslesen, lautet *getXXX*. Diese Methoden wandeln die Werte der einzelnen Spalten in entsprechende Javatypen um und liefern die Java-Werte zurück.
Der SQL-Typ *INTEGER* wird z.B. durch die Methode *getInt()* in den Javatyp int umgewandelt[79].

Zwei Methoden sind hervorzuheben:

- getString()
- getObjekt().

Alle alphanumerischen Datentypen in SQL werden mit der *getString()*-Methode in den Javatyp String umgewandelt.
Mit *getObjekt()* lassen sich alle SQL-Datentypen in eine Instanz des Typs Objekt in Java konvertieren[80].

Es folgt das entsprechend erweiterte Beispiel:

```
String URL = “jdbc:odbc:mitarbeiter”;
Statement anweisung = null;
Connection verbindung = null;
verbindung = DriverManager.getConnection (URL);
anweisung = verbindung.createStatement();
```

[79] Vgl . Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.127f
[80] Vgl. Wolff, C.: Einführung in Java, S.372ff

```
String statement = “SELECT name, mitarbeiternr FROM mitarbeiter“;
ResultSet ergebnismenge;
ergebnismenge = anweisung.executeQuery(statement);
while (ergebnismenge.next() )  {

   String mitarbeitername = ergebnismenge.getString(“name“);
   int nummer = ergebnismenge.getInt(“mitarbeiternr“);
   System.out.println (mitarbeitername+ “/t“ + nummer);
}
```

Der SELECT-Befehl wurde im Vergleich zum vorherigen Beispiel etwas abgeändert. Nun wird der ‚name' und die ‚mitarbeiternr' aus der Tabelle ‚mitarbeiter' abgefragt.
In der *while*-Schleife wird zuerst ein *String* ‚mitarbeitername' vereinbart, dem die Ergebnismenge aus dem Attribut ‚name' der Tabelle zugewiesen wird. Mit dem Attribut ‚mitarbeiternr' wird dasselbe gemacht, es wird der Integervariablen ‚nummer' zugewiesen.
Mit dem letzten Befehl werden die zugewiesenen Abfrageergebnisse am Bildschirm ausgegeben. (“/t“ bedeutet, das die beiden Ausgabeergebnissen im Abstand eines Tabulators ausgegeben werden.)

4.4.6. Schließen der Datenbankverbindung

Das Schließen einer Datenbankverbindung erfolgt mit der Methode close(). Diese Methode ist in den Schnittstellen *Connection, Statement* und *ResultSet* enthalten. Die Anwendung der *close()*-Methode ist zwar optional, weil der Garbage Collector alle nicht mehr benötigten Objekte schließt. Jedoch sollte alle nicht mehr benötigten Instanzen geschlossen werden[81].

[81] Vgl . Petkovic, D., Brüderl, M.: Java in Datenbanksystemen, S.129

4.5. Abfrage von Metadaten

In den bisher verwendeten Beispielen wurde davon ausgegangen, dass dem Programmierer die Metadaten (Daten über Daten) zur Übersetzungszeit des Programmes bekannt war. Dies ist in der Praxis i.d.R. nicht der Fall. Häufig kennt ein Programmierer bei der Erstellung einer Applikation nicht den Aufbau eines Datenbanksystems. Aus diesem Grund gibt es in JDBC zwei Klassen, mit denen man Metadaten aktueller Verbindungen abfragen kann, die *DataBaseMetaData*-Klasse und die *ResultSetMetaData*-Klasse.

DataBaseMetaData-Klasse

Mit Hilfe dieser Klasse lassen sich Metadaten über eine Datenbank abfragen. Die Instantiierung einer solcher Klasse erfolgt über die *getMetaData()*-Methode des aktuelle *Connection*-Objektes und sieht folgendermaßen aus:

```
Connection verbindung=null;
DataBaseMetaData meta = verbindung.getMetaData();
```

Insgesamt verfügt die *DataBaseMetaData*-Klasse über mehr als 130 Methoden, mit denen Informationen über sämtliche Aspekte einer Datenbank in Bezug auf ihre Metadaten eingeholt werden können[82].

ResultSetMetaData-Klasse

Mit dieser Schnittstelle lassen sich Informationen über jede ermittelte Ergebnismenge abfragen. Die Instantiierung erfolgt über die *getMetaData()*-Methode der aktuelle *ResultSet*-Instanz[83]:

```
ResultSet ergebnis=statement("SELECT name FROM mitarbeiter");
ResultSetMetaData ergebnismeta = ergebnis.getMetaData();
```

[82] Vgl. Wolff, C.: Einführung in Java, S.380
[83] Vgl. Wolff, C.: Einführung in Java, S.381

Mit den Methoden der *ResultSetMetaData*-Schnittstelle lassen sich Ergebnismengen genau beschreiben:

Methode	**Rückgabewert**
int getColumnCount()	Anzahl der Spalten
String getColumnLabel()	Spaltenüberschrift einer Spalte
String getColumnName()	Name der Spalte
int getColumnType()	Spaltentyp
String getTableName()	Name der Tabelle

(Tabelle nach Wolff, C.: Einführung in Java, S.381)

5. Remote Method Invocation (RMI)

5.1. Einleitung

Seit dem JDK (Java Development Kit) 1.1 stellt Java mit dem Paket java.rmi.* eine Möglichkeit zur Verfügung mit entfernten Objekten zu kommunizieren, mit Objekten also, die nicht auf dem lokalen Rechner vorhanden sind.

Das Prinzip, dass entfernte Objekte mit dem remote process call kommunizieren, ist schon seit längerem bekannt. In Java wird dies mit der Remote Method Invocation verwirklicht, also dem Aufruf einer entfernten Methode. Mit RMI lassen sich entfernte Objekte so handhaben, als wären sie lokale Objekte.

Um ein entferntes Objekt (remote Object) aufrufen zu können, muss es durch eine oder mehrere entfernte Schnittstellen (remote interfaces) beschrieben werden. Solche Interfaces vereinbaren die Methoden des entfernten Objektes. Mit RMI wird schließlich eine Methode des entfernten Interface aufgerufen, das von einem entfernten Objekt abgeleitet ist. Der Aufruf erfolgt dann analog zum Aufruf einer lokalen Methode.

5.2. Die Systemarchitektur von RMI

5.2.1. Das Drei-Schichten-Modell

Eine RMI-Anwendung, d.h. die Kommunikation zwischen zwei Java-Systemen, wird über drei Schichten abgewickelt. Zum einen über die Stub/Skeleton-Schicht, zum zweiten durch die Remote Reference-Schicht und schließlich mit hilfe der Transportschicht. Folgende Grafik verdeutlicht die Anordnung dieser Schichten:

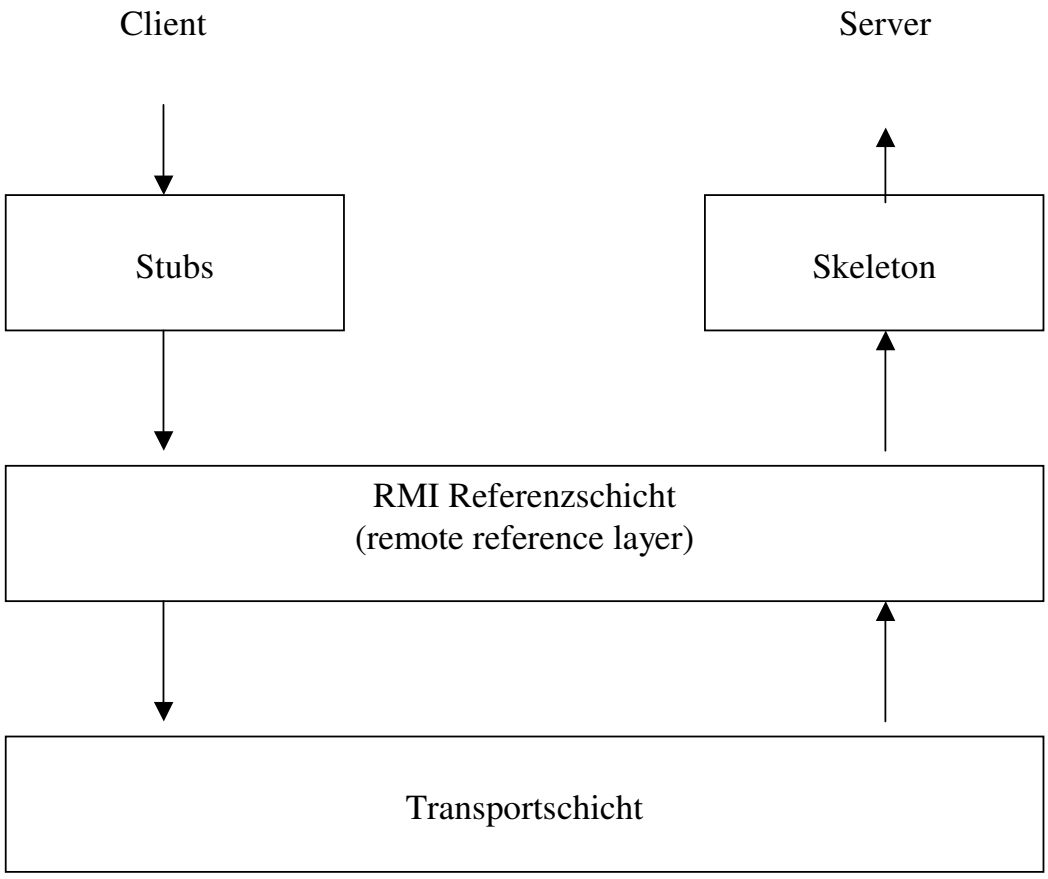


Abb.: Drei-Schichten-Modell von RMI[84]

[84] Nach Grosso, W.: Java RMI, S.159

Stub/Skeleton-Schicht

Diese Schicht ist die Verbindung zwischen den Anwendungssystemen und dem Rest des RMI-Systems.
Das entfernte Objekt muss sowohl auf der Clientseite, als auch auf der Serverseite gegenwärtig sein. Das reale Objekt, welches auf der Serverseite vorhanden ist, erzeugt eine Stubklasse. Der Client referenziert diese dann und erstellt eine Proxyklasse, also eine Platzhalter-Klasse.
Wenn der Client nun eine Methode des entfernten Objektes aufruft, so stellt die Stubklasse die Verbindung zur entfernten JVM (Java Virtual Machine) her, liest und sendet die Argumente und wartet auf die Ergebnisse, die dann dem Client zur Verfügung gestellt wird.
Die Skeleton-Klasse organisiert die Kommunikation zwischen der Stubklasse und dem implementierten Objekt. Sie liest die übergebenen Parameter der Methode ein, richtet den Aufruf an das Objekt, liest den Rückgabewert und gibt ihn an die Stubklasse ab[85].
Seit der Version 1.2. des JDK benutzt Java ein eigenes Protokoll, das Java Remote Method Protocol (JRMP), um die Kommunikation zwischen den zwei Maschinen zu organisieren. Damit ist die Skeletonklasse seit JDK 1.2 überflüssig.

Remote Referenz-Schicht

Diese Schicht bildet die eigentliche Verbindung zwischen Stub- und implementierten Objekt. Sie stellt ein *RemoteRef*-Objekt zur Verfügung auf das das Stub-Objekt die *invoke()*-Methode aufruft, wodurch letztendlich der Methodenaufruf auf dem entfernten Objekt vollzogen wird[86]. Damit ein Client auf einen entfernten Dienst eines Servers zugreifen kann, muss dieser vom Server instantiiert und an das RMI System exportiert werden.

[85] Vgl. Grosso, W.: Java RMI, S.66ff
[86] Vgl. Grosso, W.: Java RMI, S.98

Die Transportschicht

Die Transportschicht übernimmt den eigentlichen Transfer der Daten zwischen zwei Javarechnern mit Hilfe von TCP/IP (Transport Control Protocol/ Internet Protocol). TCP/IP stellt eine konsistente Verbindung her, welche über Streams die Daten übermitteln. Die Adressierung schließlich erfolgt über die IP-Adresse und einer Portnummer oder durch den DNS (Domain Name System)-Name für den Zielrechner und einer Portnummer[87].

5.2.2. Serialisierung von Objekten

Mitunter kommt es vor, dass Methoden Objekte als Parameter übergeben oder als Ergebnis zurückgeliefert werden sollen. Da die Datenübertragung eigentlich nur für primitive Datentypen funktioniert, weil die Strukturierung eines Objektes berücksichtigt werden muss, wird ein Objekt serialisiert; d.h. aus einem Objekt wird ein Datenfluss (Stream) generiert. Der Empfänger kann dadurch wieder das ursprüngliche Objekt zusammensetzen. Solches Zerlegen und Zusammensetzen eines Objektes wird als marshalling, bzw. unmarshalling bezeichnet. Diese Aufgabe übernimmt bei RMI die Stub-Klasse.
Um ein Objekt serialisierbar zu machen, muss es das Serialization Interface implementieren, das sich im java.io-Paket befindet[88].

[87] Vgl. Grosso, W.: Java RMI, S. 351ff
[88] Vgl. Grosso, W.: Java RMI, S. 70f

5.2.3. Der RMI-Namensdienst

Damit der Client überhaupt den Zielserver mit dem entfernten Objekt findet, muss vor Beginn einer RMI-Verbindung auf diesem Rechner der Namensdienst von RMI gestartet werden, die rmiregisty.

Der Server, der ein Objekt des implementierten Dienstes erzeugt, exportiert dieses an RMI. Dabei wird das Objekt in der RMI Registry registriert und an einen offiziellen Namen gebunden.

Der Client bekommt den Zugang zur RMI-Registry über die Klasse *Naming*. Dazu wird die Methode *lookup()* aufgerufen, die als Argument die IP-Adresse des RMI Registry Host-Rechners bzw. den DNS-Namen, sowie den offiziellen Namen des entfernten Dienstes benötigt. Der Rückgabewert der Methode ist dann die gewünschte Referenz auf das entfernte Objekt[89].

Die Klasse *Naming* verfügt über mehrere Methoden, mit denen die oben beschriebenen Aufgaben erfüllt werden[90]:

Methode	**Erklärung**
void bind (String einName, Remote einEnferntesObjekt)	ein entferntes Objekt wird an einen Namen gebunden
String[] list(String einName)	listet alle URLs einer registry auf
Remote lookup(String einName)	liefert das entfernte Objekt mit dem angegebenen Namen
void rebind(String einName, Remote einEntferntesObjekt)	Ein bereits mit einem Namen verbundenes entferntes Objekt erhält einen neuen Namen, alle bisherigen Bindungen werden ersetzt
void unbind (String einName)	Die Bindung eines entfernten Objektes an einen Namen wird aufgelöst.

[89] Vgl. Grosso, W.: Java RMI, S. 71ff

[90] Vgl. Wolff, C.: Einführung in Java, S. 358

5.3. Der Aufbau einer RMI-Anwendung

In diesem Unterkapitel wird eine einfache RMI-Anwendung schrittweise erzeugt. Um die Funktionalität von RMI darzustellen wird auf einem Server ein Objekt erstellt, welche nur den String „Hallo Welt“ ausgibt. Auf der Serverseite soll dieser String dann angezeigt werden.

5.3.1. Definieren einer Schnittstelle

Im ersten Schritt wird eine Schnittstelle definiert, die die gesamte Funktionalität des entfernten Objektes enthält. Es werden also die Aktionen definiert, die der Client mit dem entfernten Objekt durchführen kann. Für den Fall aus, dass nur ein String ausgegeben werden soll, sieht das Interface folgendermaßen aus:

```
public interface Hallo extends java.rmi.remote   {
String message ()  throws  java.rmi.RemoteException;
}
```

Das Interface muss zuerst als *public* definiert sein, damit der Client überhaupt darauf zugreifen kann. Da jede entfernte Schnittstelle, auf die über ein Netzwerk zugegriffen werden soll, von java.rmi.remote abgeleitet werden muss,[91] zeigt auch das gegebene Beispiel diese Abhängigkeit.
In der zweiten Zeile wird die *String*-Methode des Interfaces beschrieben. Per Definition enthalten Interfaces nur abstrakte Methoden, d.h. alle in einem Interface vereinbarten Methoden beschreiben nur den Methodenkopf, der Methodenrumpf hingegen ist leer (s.a. Kapitel 3.2.3.).
Jede Methode in einem von *Remote* abgeleiteten Interface muss eine *RemoteException* werfen[92].

[91] Vgl. Hilfe zu JBuilder4: java.rmi Remote Interface
[92] Vgl. Grosso, W.: Java RMI, S. 112

Die *RemoteException* ist eine Ausnahmebehandlung, die anzeigt, dass etwas Unvorhergesehenes im Netzwerk passiert ist. Wenn z.B. ein Server aus irgendeinem Grund zusammenbrechen sollte, erhält der Client über die *RemoteException* eine entsprechende Fehlermeldung[93].

5.3.2. Der Remote Server

In diesem Abschnitt wird eine Klasse für das entfernte Objekt geschrieben, welche das in Punkt 5.3.1. erstellte Interface implementiert. In dieser Klasse werden zudem sämtlich Methoden des Interfaces überschrieben, wobei das in dem Beispiel nur eine sein wird.
Anschließend wird in der main-Methode ein Sicherheitsmanager gestartet und ein Objekt dieser Klasse instantiiert und diese Instanz dann bei dem RMI Namensdienst bekannt gemacht.

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

public class HalloWelt extends UnicastRemoteObject
                        implements Hallo        {

public HalloWelt ()   throws RemoteException {
super();
}

public String message () throws RemoteException {
String s="Hallo Welt";
return s;
}

public static void main (String [] args){
System.setSecurityManager(new RMISecurityManager());
```

[93] Vgl. Grosso, W.: Java RMI, S. 74

```
try  {
    Hallowelt obj = new Hallowelt();
    Naming.rebind (“neuesObjekt“,obj);
    System.out.println(“Server in registry eingebunden”);
}
catch (Exception e) {
     System.out.println(“Fehler: “+ e.getMessage());
     e.printStackTrace();
}
}
```

In diesem Beispielprogramm werden zuerst die Pakete *java.rmi.** und *java.rmi.server.UnicastRemoteObject* importiert, so dass dem Programmierer sämtliche Klassen dieser Pakete zur Verfügung stehen.

Nach den Importanweisungen wird im Klassenkopf eine *public*-Klasse mit Namen Hallowelt vereinbart, die von der Klasse *UnicastRemoteObject* abgeleitet ist und das Interface Hallo implementiert hat.

Die Klasse wird abgeleitet von der *UnicastRemoteObject*-Klasse, weil sie damit RMI bekannt gemacht wird. Eine *UnicastRemoteObject*-Klasse benutzt einen TCP-Strom, um Daten zu übermitteln und ist nur für die Dauer des Serverprozesses aktiv[94].

Als nächstes wird der Konstruktor definiert. Er hat eine *RemoteException*, deren Aufgabe oben beschrieben wurde. Der Konstruktor der Klasse Hallowelt ruft mit Hilfe des Schlüsselwortes *super()* den Konstruktor der Oberklasse auf, damit eine Instanz dieser Klasse entsteht.

[94] Vgl. Grosso, W.: Java RMI, S. 137f

Da das Interface Hallo implementiert wurde, ist es zwingend notwendig, alle Methoden des Interfaces zu überschreiben, weil dort nur die Methodenköpfe definiert werden, aber die Methodenrümpfe leer sind. Dieser Vorgang wird mit dem vorgegebenen Anweisungsblock ausgeführt. Da die Methode überschrieben wird, hat sie denselben Namen wie im Interface. In der Methode wird ein String vereinbart, dem die Worte „Hallo Welt“ zugeordnet werden, und er wird dann mit dem Schlüsselwort *return* der Methode zurückgegeben.

Im nächsten Schritt wird die main-Methode vereinbart. In Java wird eine Applikation immer durch eine main-Methode gestartet.
Zuerst wird eine neue Instanz der *RMISecurityManager*-Klasse gebildet. Ohne eine solche Instanz lädt RMI keine Klassen von dem entfernten Objekt[95].

Am Ende der Server-Applikation wird ein try-catch-Block durchlaufen. Dieser wird geschrieben, um im try-Teil einen oder mehrere Befehle zu testen. Sollte es zu einem Fehler kommen, wird er im catch-Block abgefangen. Tritt kein Fehler auf, wird der catch-Block übersprungen.
Im try-Block wird zuerst nach der üblichen Art und Weise eine Instanz der Klasse Hallowelt erzeugt. Anschließend wird die Methode *rebind* der Naming-Klasse aufgerufen, mit der diese Instanz in den RMI-Namensdienst, unter dem Namen „neuesObjekt“, verankert wird. Der Name kann auch eine URL-Adresse aus dem Internet sein.
Die letzte Anweisung gibt den ihr zugewiesenen Text aus, nachdem das Objekt erfolgreich instantiiert und an den RMI-Namensdienst angebunden wurde. Sollte dabei ein Fehler auftreten, springt das Programm direkt zu dem catch-Block.

Im Kopf des catch-Blocks erhält die catch-Anweisung eine benannte Exception als Parameter. Im Rumpf des Blockes wird das Wort „Fehler“ ausgegeben und mit der Methode *e.getMessage*() wird die Art des Fehlers vom System ausgegeben.
Mit *e.printStackTrace*() wird der Fehlers einem Output-Stream übergeben, damit es angezeigt werden kann.

[95] Vgl. Grosso, W.: Java RMI, S. 453

5.3.3. Die Client-Seite

Auf der Client-Seite wird in der main-Methode eine Referenz auf das entfernte Objekt erstellt und diese Methode wird ausgeführt.

```
import java.rmi.* ;

public class Client {
  public static void main (String [] args) {
    try  {
      Hallo neu = (Hallo) Naming.looup(“rmi:// neuesObjekt”);
      String s=neu.message();
      Sytem.out.println(s);
      }
    catch (Exception e) {
       System.out.println(“Fehler: “+ e.getMessage());
        e.printStackTrace();
      }
}
```

Die Importanweisung und die Klassendeklaration sind selbsterklärend. In der main-Methode wird wiederum, wie auf der Serverseite, ein try-catch-Block aufgebaut, dessen Funktionsweise oben schon erklärt wurde.

Im try-Block wird nun eine Instanz namens neu des Interfaces Hallo gebildet, welches in die Klasse Hallowelt implementiert ist. Diese Instanz wird von der Methode *lookup()* der Klasse Naming erstellt. Damit der Client auf ein entferntes Objekt zugreifen kann, muss er dessen Namen beim RMI-Namensdienst kennen.
Bei der nächsten Anweisung im try-Block wird ein String definiert, der mit der Methode *message()*, initialisiert wird. (Dem String wird der Text „Hallo Welt“ zugewiesen). Mit dem letzten Befehl dieses Blockes wird der String ausgegeben.

Der Catch-Block hat dieselbe Aufgabe wie auf der Serverseite.

5.3.4. Starten des RMI-Systems

Um das System zu starten, müssen selbstverständlich alle Klassen kompiliert werden.
Die Stub- und Skeletonklassen werden nach folgender Syntax durch den RMI-Compiler erzeugt[96]:

```
rmic [voller Klassenname inklusive dem Packet];
```

In dem Beispiel sähe die Erzeugung der Stub- und Skeletonklassen wie folgt aus:

```
rmic Hallowelt;
```

Dieser Aufruf generiert im Verzeichnis von Hallowelt zwei Klassen:

```
Hallowelt_Skel.class
Hallowelt_Stub.class
```

Anschließend muss der RMI-Namensdienst bzw. der RMI-registry, gestartet werden. Dies geschieht mit diesem Befehl[97]:

```
Start rmiregistry;
```

Die Registry belegt standardmäßig den Port 1099. Sollte ein anderer Port belegt werden, so lautet die Syntax:

```
Rmiregistry portnummer &;
```

Auf dem Rechner, auf dem die RMI-registry läuft, wird auch der Server gestartet, der über die Registry seien Dienst zur Verfügung stellt.
Der Client kann auf irgendeinem anderen Rechner arbeiten, der mit dem Server verbunden ist, und die Dienste des Servers in Anspruch nehmen.

---

[96] Vgl. Grosso, W.: Java RMI, S. 68
[97] Vgl. Hilfe zu JBuilder4: rmiregistry-The Java Remote Object Registry

6. Zusammenfassung

Die Datenbankanbindung über Java wird sich aufgrund seiner Vorteile auch in Zukunft sowohl im privatem, als auch im kommerziellen Bereich ihren angemessenen Stellenwert behaupten und ausbauen.

Mit JDBC ist dem Programmierer ein Entwicklungswerkzeug an die Hand gegeben worden, mit dem er herstellerunabhängig Datenbankverbindungen, relativ leicht erlernbar, erstellen kann. Das haben auch die Anbieter von kommerziellen Datenbanksystemen erkannt, was an dem steigenden Angebot an reinen Java-Treibern (Typ-4-Treiber) für diverse Datenbanksysteme erkennbar ist.
Als zweiten grossen Vorteil, neben der relativen leichten Programmierbarkeit, ist die Plattformunabhängigkeit von Java zu erwähnen. Mit diesem Prinzip kann ein Javaprogramm auf jedem beliebigen System, das über eine Java Virtual Machine verfügt, ausgeführt werden.

Der grosse Nachteil von Java ist die schlechte Perfomance bei Datenbankzugriffen, welche sich allerdings meist nur bei zeitkritischen Anwendungen äussert[98].

Vergleicht man die Vorteilen mit den Nachteilen, so ist JDBC, besonders in Verbindung mit RMI, ein mächtiges Instrument zum konstruieren verteilter Anwendungen mit Datenbankzugriffen, so überwiegen eindeutig die Vorteile.
Der Perfomancenachteil kann durch ein entsprechendes Design des Datenbanksystems und durch leistungsstarke Server verringern werden. Des weiteren zeigt die Entwicklung der Computerhardware der letzten Jahre, dass Rechner immer leistungsstärker werden, bei gleichbleibenden, bzw. sinkenden Preisen, womit der Perfomancenachteil immer weniger ins Gewicht fällt.

[98] Zur Perfomance bei JDBC: Petkovic, d., Brüderl, M.: Java in Datenbanksystemen, Kapitel 6

Literaturverzeichnis

Ebner, Michael: SQL lernen, München, Addison-Wesley-Longman GmbH, 1999

Fink, Andreas; Schneidereit, Gabriele; Voß, Stefan: Grundlagen der Wirtschaftsinformatik, Heidelberg, Physika-Verlag, 2001

Flanagan, David: Java in a nutshell, Deutsche Ausgabe für Java 1.1, 2.Auflage, Sebastopol, USA, O'Reill, 1998

[Hrsg.] Disterer, Georg; Fels, Fiedrich; Hausotter, Andreas: Taschenbuch der Wirtschaftsinformatik, 2.Auflage, München, Fachbuchverlag Leipzig, 2003

Grosso, William : Java RMI, 1. Auflage, Sebastopol, USA, O'Reilly, 2002

Hartwig, Jens: Einführung in JDBC, unter http://web.f4.fhtw-berlin.de/hartwig /JDBC/jdbc.html, 1997

Heuer, Andreas; Saake, Günter; Settler, Kai. Uwe: Datenbanken kompakt, 1. Auflage, Bonn, mitp-Verlag, 2001

Louis, Dirk ; Müller, Peter: Jetzt lerne ich Borland JBuilder4, München, Markt+Technik-Verlag, 2001

Louis, Dirk ; Müller, Peter: Jetzt lerne ich Java, München, Markt+Technik-Verlag, 2000

Matthiesen, Günter; Unterstein, Michael: Relationale Datenbanken und SQL, 2.Auflage, München, Addison-Wesley-Verlag, 2000

Petkovic, Dusan; Brüderl, Markus: Java in Datenbanksystemen JDBC, SQLJ, Java DB-Systeme und -Objekte, München, Addison-Wesley-Verlag, 2002

Rautenstrauch, Claus; Schulze, Thomas: Informatik für Wirtschaftswissenschaftler und Wrtschafts-Informatiiker, Berlin, Heidelberg, Springer-Verlag, 2003

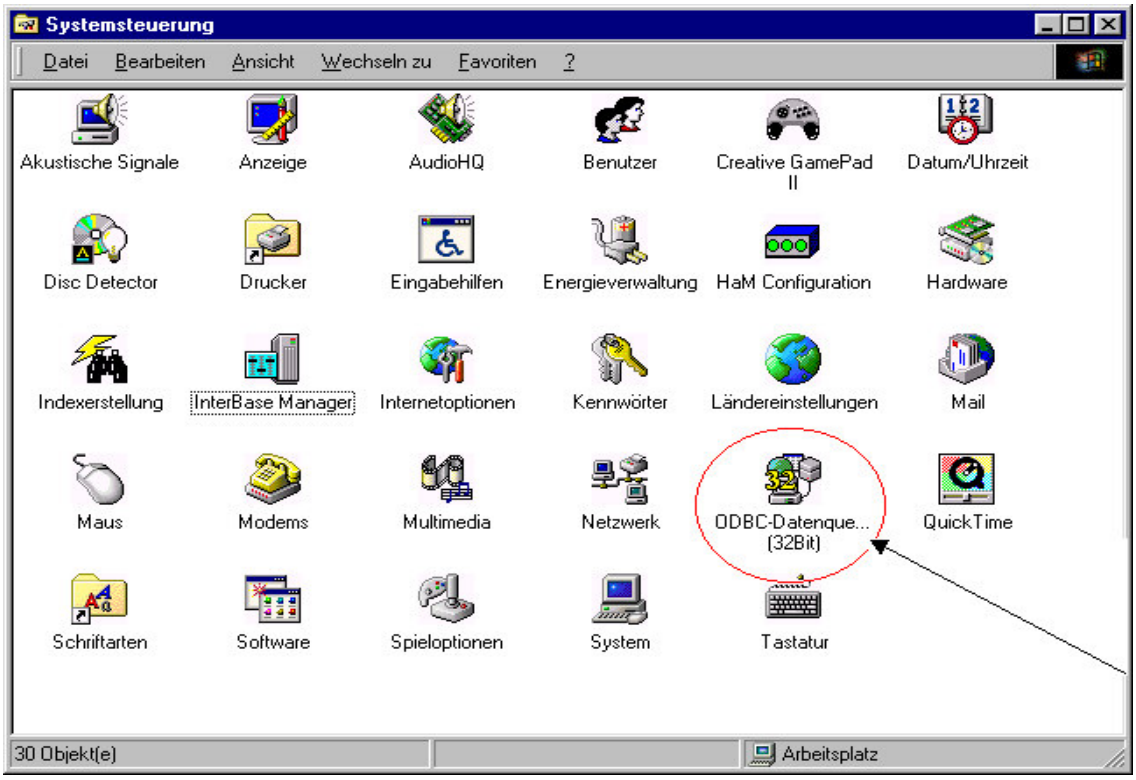
Wolff, Christian: Einführung in Java Objektorientieres Programmieren mit der Java 2-Plattform, Leipzig, B.G.Teubner Stuttgart 1999

Anhang

Das vorliegende Programm wurde von mir mit Hilfe des JBuilders4, ohne Zuhilfenahme des Designers, im Rahmen meiner Diplomarbeit „Datenbank- und Netzwerkprogrammierung in Java“ geschrieben. Das Progamm kann Datenbanken über einen JDBC/ODBC-Treiber anbinden. Um Datenbanken mit dieser Methode anzubinden, müssen sie über die Systemsteuerung als ODBC-Datenquelle bekannt gemacht werden. Dieser Vorgang wird nun erläutert. Ziel ist eine Access-Datenbank als ODBC-Datenquelle bekannt zu machen. Auf der der Diplomarbeit beiliegenden CD-ROM sind zwei von mir in Access erstellte Beispieldatenbanken („europa“ und „mitarbeiter“) beigefügt.

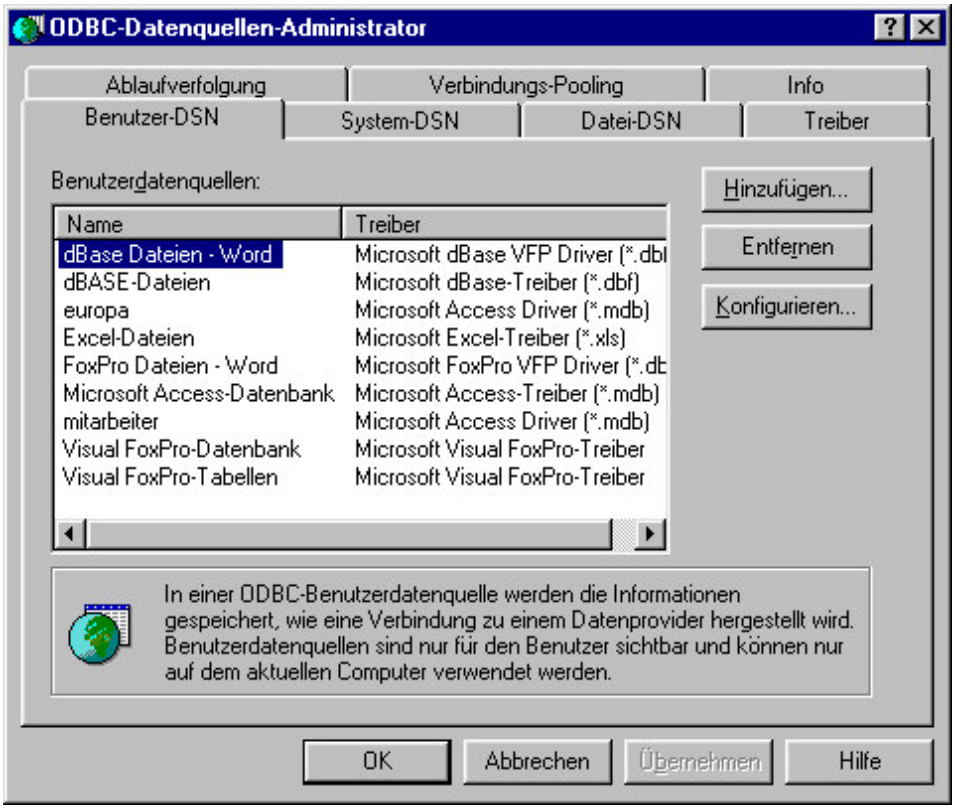
1.: Der Ordner Systemsteuerung wird über das Startmenü/Einstellungen geöffnet

2. : Das Symbol „ODBC-Datenquellen (32Bit)“ doppelklicken.



3.: Die Schaltfläche „Benutzer DSN“ aktivieren.

4.: Auf „Hinzufügen“ klicken



5.: Den Treiber auswählen für eine Acceess-Datenbank auswählen (Microsoft Access Driver(*.mdb)). Wird der Treiber nicht aufgeführt, ist er nicht korrekt unter Windows installiert.

6.: Bei „Datenquellenname“ wird der Name der neuen Treiberverbindung eingegeben. Mit der Schaltfläche „Auswählen“ wird festgelegt, auf welche Datenbank die Treiber-verbindung zugreifen soll.
Alle anderen Angaben sind optional. Es ist davon abzuraten der Datenbanken eine Kennung und ein Kennwort zu geben, da das Programm nur mit Datenbanken ohne Kennung und Kennwort läuft. Damit das Programm einwandfrei läuft, sollte der der Datenquellenname mit dem Namen der Tabelle, aus der Daten ausgelesen werden soll, der Datenbank übereinstimmen. Die Beispieldatenbanken haben nur jeweils eine Tabelle, deren Name mit dem der Datenbank übereinstimmt.

7.: Mit „OK“ das Fenster schließen und alle anderen auch.

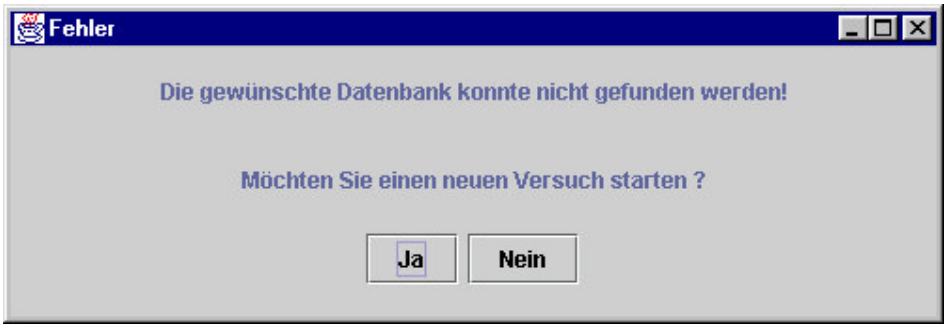
Wurden die Datenbanken dem ODBC-Treiber bekannt gemacht, kann das Programm über den JBuilder4 gestartet werden. Dabei ist zu beachten, dass alle Ordner und Dateien in das übliche Verzeichnis kopiert wurden. Dann kann der Quelltext des Programms über „Datei“ und „Projekt öffnen“ aus dem Ordner, in dem die Dateien und Ordner gespeichert wurden, angezeigt werden.

Erläuterung zum Programm

Die einzelnen Fenster, die angezeigt werden sind zumeist selbsterklärend. Dennoch werden sie hier kurz erläutert.
Das Programm wird mit der Klasse „Start“ gestartet. Nach dem Programmstart öffnet sich das erste Fenster:

Der Name der Datenbank, aus der Daten abgefragt werden soll, wird hier eingegeben.

Sollte eine Datenbank nicht gefunden werden, wird folgendes Fenster angezeigt:



‚Ja’: Das erste Fenster wird wieder angezeigt
‚Nein’: Programmende.

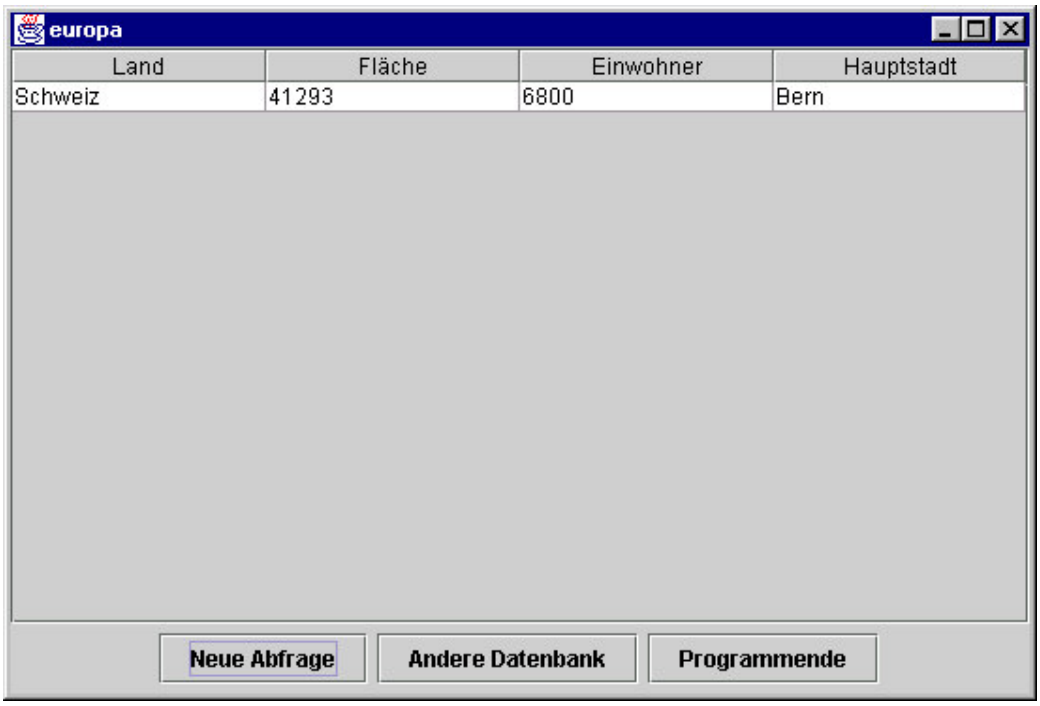
Wird eine Datenbank gefunden, erscheint das 2.Fenster:

Rechts oben wird ein Attribut ausgewählt. Darunter wird dann ein Suchstring eingegeben (z.B. Schweiz).
Wird das Häckchen bei „Alle Datensätze anzeigen“ aktiviert, werden alle Datensätze angezeigt und eine mögliche Eingabe in dem oberen Feld ignoriert.

Sollte eine Suche nach einem Datensatz erfolglos gewesen sein wird das folgende Fenster angezeigt:

‚Neue Abfrage’: Das 2. Fenster wird wieder geöffnet.

Wurde ein oder mehrere Datensätze gefunden wird dieses Fenster aufgebaut:



‚Neue Abfrage': Das 2. Fenster wird geöffnet

‚Andere Datenbank': Das 1. Fenster wird geöffnet

Der Quelltext des Programms

Die Klasse Start

```
package jdbcodbcbridge;

public class Start
{
   // Variablenvereinbarung

   static Frame1 Fenster;

   // main-Methode

   public static void main(String[] args)
   {
     Fenster = new Frame1("Abfrage");
     Fenster.pack();
     Fenster.setBounds(300,300,300,130);
     Fenster.show();

   }

   // leerer Konstruktor (wird von "Frame1" aufgerufen)

   public Start ()
   {
   }

   /**Konstruktor zum Aufrufen des Frame1,
     wird von Frame3 und Frame4 aufgerufen */

   public Start (String titell)
   {
     Fenster = new Frame1("Abfrage");
     Fenster.pack();
     Fenster.setBounds(300,300,300,130);
     Fenster.show();
   }

   // Methode, die die Instanz 'Fenster' schliesst

   public void schliessen ()
   {
      Fenster.dispose();
   }

}
```

Die Klasse Frame1

```
package jdbcodbcbridge;


//  Importieranweisungen

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.sql.*;


public class Frame1 extends JFrame
{
    // Variablenvereinbarungen

    Statement bef = null;
    Connection verb = null;

    JTextField text1;
    String name;
    int vergleich;

    Start start=new Start();

    static Frame4 Fenster4;
    static Frame2 Fenster2;

  // Konstruktor, der von "Start" aufgerufen wird

  public Frame1(String titel)
  {

   // Aufruf des Konstruktors der Oberklasse

   super(titel);

   // Gestaltung des Fensters

   JButton knopf = new JButton("Ok");
   JLabel label1 = new JLabel("Bitte Name der Datenbank eingeben",0);
   text1=new JTextField();

   JPanel panel1 = new JPanel();
   panel1.setLayout(new FlowLayout());
   panel1.add(knopf);
```

```
  getContentPane().setLayout(new GridLayout(3,1));
  getContentPane().add(label1);
  getContentPane().add(text1);
  getContentPane().add(panel1);

  //'Listener' werden angefügt
  knopf.addActionListener(new Lauscher());
  addWindowListener(new WindowLauscher());

  }


  // Konsttruktor, der von "Frame3" aufgerufen wird

public Frame1(String titel, java.sql.Statement statem, java.sql.Connection con)
{
    Fenster2 = new Frame2(titel,statem,con);
    Fenster2.pack();
    Fenster2.setBounds(225,300,550,180);
    Fenster2.show();

}

// innere Klasse, die einen ActionListener definiert

class Lauscher implements ActionListener
{

    public void actionPerformed(ActionEvent e)
   {

    /** Text des JTextField wird eingelesen und dem String 'url'
      wird der Name der DB-Verbindung übergeben*/

    name = text1.getText();
    String  url = "jdbc:odbc:"+name;

    // Treiber wird geladen

       try
       {
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       }

       catch (Exception f){

       System.out.println("Treiber konnte nicht geladen werden");
       return;
       }
```

```
    /** DB-Verbindung wird hergestellt (Die System.out.println()-Anweisung ist optional)
        es wird ein sensitiver Cursor für die ResultSet-Schnittstelle vereinbart*/

       try
       {

       verb = DriverManager.getConnection (url,null,null);
       System.out.println("Es konnte eine Verbindung hergestellt werden");
       bef = verb.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
             ResultSet.CONCUR_READ_ONLY);

       }


       catch (Exception f)
       {
       System.out.println("Es konnte keine Verbindung hergestellt werden");
       vergleich = 1;
       }

       /** Verzweigung: Konnte keine Verbindung hergestellt werden wird Frame4
       aufgerufen, sonst Frame2*/

       if(vergleich == 1)
       {
       //das aktuelle Fenster wird geschlossen

       start.schliessen();
       //ein neues Fenster wird aufgebaut
       Fenster4 = new Frame4("Fehler");
       Fenster4.pack();
       Fenster4.setBounds(250,300,500,170);
       Fenster4.show();

       }

       else
       {
       //das aktuelle Fenster wird geschlossen

       start.schliessen();
       // neues Fenster wird aufgebaut
       Fenster2 = new Frame2(name,bef,verb);
       Fenster2.pack();
       Fenster2.setBounds(225,300,550,180);
       Fenster2.show();

       }
   }

}
```

```
// innere Klasse, die bei Schliessen des Fensters aktiviert wird

class WindowLauscher extends WindowAdapter
{

  public void windowClosing(WindowEvent e)
  {
    // schliessen der DB-Verbindung

    try
    {
     bef.close();
     verb.close();
    }
    catch(Exception g){


    }

    // JVM wird geschlossen

    System.exit(0);
  }
}

//leerer Konstruktor, der von "Frame2" und "Frame4" aufgerufen wird

public Frame1()
{
}

//zwei Methoden zum Schliessen der jeweiligen Instanzen

public void frame4schliessen (){

Fenster4.dispose();
}

 public void frame2schliessen (){

 Fenster2.dispose();
}

}
```

Die Klasse Frame2

```
package jdbcodbcbridge;

// Importanweisungen

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.sql.*;


public class Frame2 extends JFrame
{

   // Varialen und Instanzen

   Statement befehl = null;
   Connection verbindung = null;

   ResultSet rs;
   ResultSetMetaData rsmd;

   int i,j;
   boolean b;

   JCheckBox check;
   JTextField text1;
   JComboBox box;
   JTable tabelle;

   String name;
   String eingabe;
   String auswahl;
   String[] spaltennamen;
   String[][] ausgabe;

   Frame1 frame=new Frame1();
   static Frame3 Fenster3;


   //Konstruktor, der von "Frame1" aufgerufen wird

  public Frame2 (String titel, java.sql.Statement statement1, java.sql.Connection con)
   {
   // Aufruf des Konstruktors der Oberklasse

    super("Datenbank: "+titel);
```

```
// Übergabeparameter werden in den Klassenvariablen geschrieben

befehl=statement1;
verbindung=con;
name=titel;

// Gestaltung des Fensters

box = new JComboBox();
JButton button = new JButton("Ok");
check = new JCheckBox("Alle Datensätze anzeigen");
JLabel label1 = new JLabel("Bitte das gewünschte Attribut markieren: ");
JLabel label2 = new JLabel("Bitte den Suchstring angeben: ");
text1 = new JTextField();

JPanel panel1=new JPanel();
JPanel panel2=new JPanel();
JPanel panel3=new JPanel();
JPanel panel4=new JPanel();

panel1.setLayout(new GridLayout(1,2));
panel2.setLayout(new GridLayout(1,2));
panel3.setLayout(new FlowLayout());
panel4.setLayout(new FlowLayout());

panel1.add(label1);
panel1.add(box);
panel2.add(label2);
panel2.add(text1);
panel3.add(check);
panel4.add(button);

getContentPane().setLayout(new GridLayout(4,1));
getContentPane().add(panel1);
getContentPane().add(panel2);
getContentPane().add(panel3);
getContentPane().add(panel4);

// Zuweisung der Listener

button.addActionListener(new lauscher());
addWindowListener(new WindowLauscher());
```

```
try
    {
     /**An die Instanz von ResultSet, 'rs', werden alle Datensätze
       der DB übergeben*/

     rs = befehl.executeQuery("SELECT * FROM "+titel);
     //rsmd erhält die Metadaten der DB
     rsmd = rs.getMetaData();

     // Ermittlung der Anzahl der Spalten in der DB

     i=rsmd.getColumnCount();

     spaltennamen = new String [i];

     /**Die Bezeichnungen der einzelnen Spalten werden dem Srring übergeben
      und diese der JComboBox übergeben*/

     for (int y=1; y<=i; y++)
     {
     spaltennamen[y-1]=rsmd.getColumnName(y);
     box.addItem(spaltennamen[y-1]);
     }
    }


   catch (Exception e)
   {
   e.printStackTrace();
   }

}

// Innere Klasse

class lauscher implements ActionListener {

    public void actionPerformed (ActionEvent e)
    {

     try
     {

      // hier wird geprüft, ob das Häckchen in der JCheckBox aktiviert ist

      b=check.isSelected();
```

```
if (b==true)
{

/**der Cursor der Ergebnismenge springt an die letzte Stelle und gibt
die Nummer mit getRow() an die Variable i
(Ermittlung der Anzahl der Zeilen)*/

rs.last();
j=rs.getRow();

// Cursor wird wieder vor die erste Zeile gesetzt

rs.beforeFirst();

//Bildung eines Strings mit der Anzahl der ermittelnden Zeilen und Spalten

ausgabe=new String [j][i];

// Einlesen der Datenfelder in den String

while(rs.next())
{
  int x=rs.getRow();

  for(int y=1;y<=i;y++)
  {
    ausgabe[x-1][y-1]=rs.getString(y);
  }
}
// Bildung der Tabelle durch Übergabe des Strings und der Spaltennamen

tabelle=new JTable(ausgabe,spaltennamen);

//aktuelles Fenster wird geschlossen

frame.frame2schliessen();

//neues Fenster wird instantiiert

Fenster3=new Frame3(name,tabelle,befehl,verbindung);
Fenster3.pack();
Fenster3.setBounds(200,200,550,370);
Fenster3.show();

}
```

```
 //wenn das Häckchen nicht aktiviert ist:

else
{
 // Der Text des JTextField wird eingelesen

 eingabe=text1.getText();

 // Das ausgewählte Item der JComboBox wird eingelesen

 auswahl=(box.getSelectedItem()).toString();

 String a;

 // Die Nummer des ausgewählten Item wird ermittelt

 int stelle=box.getSelectedIndex();

 // Der Klassentyp des ausgewählten Items wird ermittelt

 String typ=rsmd.getColumnClassName(stelle+1);

 /**Ist der ermittelte Klassentyp vom Typ 'String', so wird in der
     WHERE-Klausel mit Hochkommata abgefragt, sonst ohne. */

 if (typ.equals("java.lang.String"))
 {
   a="SELECT * FROM "+name+" WHERE "+auswahl+"='"+eingabe+"'";
 }

 else
 {
   a="SELECT * FROM "+name+" WHERE "+auswahl+"="+eingabe;
 }

/**An die Instanz von ResultSet, 'rs', werden alle ermittelnden Datensätze
 der DB übergeben*/

rs=befehl.executeQuery(a);

//Ermittlung der Anzahl der Reihen in der Ergebnismenge

rs.last();
j=rs.getRow();
rs.beforeFirst();
```

```
  // wenn Datensätze in der Ergebnismenge:

    if (j!=0)
    {

      ausgabe=new String [j][i];
      while(rs.next())
      {
        int x=rs.getRow();

         for(int y=1;y<=i;y++)
         {
           ausgabe[x-1][y-1]=rs.getString(y);
         }
       }

    tabelle=new JTable(ausgabe,spaltennamen);

    // schliessen des aktuellen Fensters

    frame.frame2schliessen();

    // neues Fenster wird gebildet

    Fenster3=new Frame3(name,tabelle,befehl,verbindung);
    Fenster3.pack();
    Fenster3.setBounds(300,300,550,370);
    Fenster3.show();
    }

    // wenn keine Datensätze gefunden wurden

    else
    {
     // Fenster schliessen
     frame.frame2schliessen();

     // neues Fenster wird gebildet

     Fenster3 = new Frame3("Keine Datensätze gefunden",
                 name,auswahl,eingabe,befehl,verbindung);
     Fenster3.pack();
     Fenster3.setBounds(300,300,550,145);
     Fenster3.show();

    }
  }
}
```

```
    // Ausnahmebehandlung

    catch (Exception g)
    {

     frame.frame2schliessen();

     Fenster3 = new Frame3("Keine Datensätze gefunden",
                name,auswahl,eingabe,befehl,verbindung);
     Fenster3.pack();
     Fenster3.setBounds(200,200,550,145);
     Fenster3.show();

    }
  }
}

 // Innere Klasse, die bei Schliessen des Fensters aktiviert wird

 class WindowLauscher extends WindowAdapter
  {

   public void windowClosing(WindowEvent e){

    // Schliessen der DB-Verbindung

     try
     {
      befehl.close();
      verbindung.close();
     }

     catch (Exception f)
     {
      f.printStackTrace();
     }

    System.exit(0);
   }


 }
  // leerer Konstruktor

   public Frame2()
   {
   }
```

```
    // Methode zum Schliessen des instantiierten Fensters

    public void schliessen()
    {
    Fenster3.dispose();
    }
}
```

Die Klasse Frame3

```
package jdbcodbcbridge;

// Importanweisungen

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.sql.*;


public class Frame3 extends JFrame
{
   // Variable und Instanzen

   Statement statement1 = null;
   Connection verb1 = null;

   String dbname;

   Frame2 Fenster2=new Frame2();

   JTable tab;

   /**Konstruktor, der von 'Frame2' aufgerufen wird
        (wenn Datensätze ausgegeben wurden )*/

  public Frame3 (String titel,javax.swing.JTable tabelle,
              java.sql.Statement statem, java.sql.Connection con)
 {
     // übliche Vorgehensweise

      super(titel);

      dbname=titel;
      statement1=statem;
      verb1=con;

      tab=tabelle;
      //Tabellenelemente lassen sich nicht editieren
      tab.setEnabled(false);

      JButton button1 = new JButton("Neue Abfrage");
      JButton button2 = new JButton("Andere Datenbank");
      JButton button3 = new JButton("Programmende");
```

```
    JPanel panel1=new JPanel();
    panel1.setLayout(new FlowLayout());

    panel1.add(button1);
    panel1.add(button2);
    panel1.add(button3);

    JScrollPane pane1=new JScrollPane(tab);

    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(pane1,BorderLayout.CENTER);
    getContentPane().add(panel1,BorderLayout.SOUTH);

    button1.addActionListener(new lauscher1());
    button2.addActionListener(new lauscher2());
    button3.addActionListener(new lauscher3());

    addWindowListener(new WindowLauscher());
}


/** Konstruktor, der von 'Frame2' aufgerufen wird
   (wenn keine Datensätze gefunden wurden*/

public Frame3 (String titel,String datenbank,String attr, String ausw,
                java.sql.Statement statem,java.sql.Connection con )
{
    // übliche Vorgehensweise

    super(titel);

    dbname=datenbank;
    statement1=statem;
    verb1=con;

    JLabel lab1 = new JLabel("An der Stelle "+ausw+" des Attributs "+attr,0);
    JLabel lab2 = new JLabel(" der Datenbank "+datenbank+" wurde leider nichts
                                gefunden",0);

    JButton button3 = new JButton("Neue Abfrage");
    JButton button4 = new JButton("Programmende");

    JPanel panel2 = new JPanel();
    panel2.setLayout(new FlowLayout());
    panel2.add(button3);
    panel2.add(button4);

    getContentPane().setLayout(new GridLayout(3,1));
    getContentPane().add(lab1);
    getContentPane().add(lab2);
    getContentPane().add(panel2);
```

```
        button3.addActionListener(new lauscher1());
        button4.addActionListener(new lauscher3());
        addWindowListener(new WindowLauscher());
    }

  // Innere Klasse

    class lauscher1 implements ActionListener
    {
      public void actionPerformed (ActionEvent e)
      {
        // Fenster schliessen

        Fenster2.schliessen();

        // neues Fenster aufrufen

        Frame1 Fenster1=new Frame1(dbname,statement1,verb1);
      }

    }


  //Innere Klasse

    class lauscher2 implements ActionListener
    {

       public void actionPerformed (ActionEvent f)
       {
        //Fenster schliessen

        Fenster2.schliessen();

        //neues Fenster aufrufen

        Start start=new Start("Abfrage");
       }

    }
```

```
// Innere Klasse

  class lauscher3 implements ActionListener
  {

     public void actionPerformed (ActionEvent g)
     {
      //schliessen der DB-Verbindung

      try
       {
        verb1.close();
        statement1.close();
       }

      catch (Exception f)
       {

        f.printStackTrace();
       }

      System.exit(0);
     }

  }

    // Innere Klasse zum Schliessen des Fensters und Beenden der JVM

    class WindowLauscher extends WindowAdapter
    {

       public void windowClosing(WindowEvent e)
       {

        try
         {
          verb1.close();
          statement1.close();
         }

        catch (Exception g)
        {

         g.printStackTrace();
         }

       System.exit(0);
       }
    }
}
```

Die Klasse Frame4

```
package jdbcodbcbridge;


// Importanweisungen

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Frame4 extends JFrame
{
   // Instanz von 'Frame1' bilden

   Frame1 neu= new Frame1();


   public Frame4(String titel)
   {
      // übliche Vorgehensweise

      super (titel);

      JButton button1 = new JButton ("Ja");
      JButton button2 = new JButton ("Nein");

      JLabel label1 = new JLabel("Die gewünschte Datenbank konnte nicht gefunden
                                  werden!",0);
      JLabel label2 = new JLabel("Möchten Sie einen neuen Versuch starten ?",0);

      JPanel panel1= new JPanel();
      panel1.setLayout(new FlowLayout());
      panel1.add(button1);
      panel1.add(button2);

      getContentPane().setLayout(new GridLayout(3,1));
      getContentPane().add(label1);
      getContentPane().add(label2);
      getContentPane().add(panel1);

      button1.addActionListener(new lauscher1());
      button2.addActionListener(new lauscher2());

      addWindowListener(new WindowLauscher());
   }
```

```
// Innere Klasse

  class lauscher1 implements ActionListener
  {

        public void actionPerformed (ActionEvent e)
        {
          // schliessen des aktuellen Fensters

          neu.frame4schliessen();

          // neues Fenster

          Start Fenster = new Start("Abfrage");
        }
   }

 // Innere Klasse zum Verlassen des Systems

 class lauscher2 implements ActionListener
 {

       public void actionPerformed (ActionEvent e)
       {
         System.exit(0);
       }
 }

// Innere Klasse zum Schliessen des Fensters und Beenden der JVM

 class WindowLauscher extends WindowAdapter
 {

        public void windowClosing(WindowEvent e)
        {
          System.exit(0);
        }


 }

}
```