

▲ Vorlesung DB-Anwendungen

▼ 1 Schnittstellen und Einbettungstechniken

▼ 1.1 Java Database Connectivity

▼ 1.2 Enterprise Java Beans

▼ 1.3 Java Data Objects

▼ 2 Architekturmuster and Umsetzungstechniken

▼ 2.1 Domänenlogik

▼ 2.2 Datenzugriff

▼ 2.3 Objekt-Relational-Abbildung und -Interoperabilität

▼ 3 Konnektivität und Offline-Techniken

▼ 3.1 XML-Strukturen

▼ 3.2 XML-Schema

▼ 3.3 XSL-Transformationen

▼ 3.4 XML-Programmierschnittstellen

▼ 3.5 XML und Datenbanken

▼ 3.6 Web Services

▲ 1 Zugriffsschnittstellen

Dieses Kapitel führt in drei Zugriffsschnittstellen auf Datenbanken und persistente Objektspeicher ein. Die Darstellung skizziert daher zunächst die auf das relationale Speicherungs- und Zugriffssparadigma ausgelegte JDBC-Schnittstelle.

Davon ausgehend wird die Kapselung von JDBC-basierter Persistenzlogik durch Enterprise Java Beans entwickelt. Hierbei steht die Ausprägungsform der bean managed persistence im Vordergrund, da sie die weitestgehenden Eingriffsmöglichkeiten für den Programmierer bietet. Abschließend wird mit den *Java Data Objects* ein jüngerer Ansatz zur Realisierung transparenter Speicherung eingeführt, der gleichzeitig verschiedenste Persistenzdienstleister unterstützt.

1.1 Java Database Connectivity

Motivation

Häufig besteht der Wunsch oder die Notwendigkeit auf bereits vorliegende Datenbestände, die durch ein Datenbankmanagementsystem (DBMS) verwaltet werden, in einer Applikationsprogrammiersprache zuzugreifen. Dabei soll die Anbindung der benötigten Datenquelle nicht problemspezifisch wieder und wieder neu entwickelt werden, sondern sollte sich auf ähnliche Datenanbindungsprobleme übertragen lassen.

Vor diesem Hintergrund liegt es nahe sich an den Typen der verfügbaren und kommerziell bedeutsamen DBMS zu orientieren und herstellerspezifische Entwicklungen außer Acht zu lassen. Gleichzeitig offenbaren sich hierbei Standardisierungsbemühungen wie die Sprache SQL zum Zugriff auf relationale DBMS als lohnenswerter Ansatz der Etablierung einer generischen und übertragbaren Schnittstelle.

Die Idee zur Schaffung einer solchen generischen Schnittstelle für den Zugriff auf relationale DBMS geht zurück auf eine Initiative der *SQL Access Group*, welche später in der Vereinigung mit der *X/Open Group* aufging, die zwischenzeitlich in *Open Group* umbenannt wurde. Das dort konzipierte programmiersprachenunabhängige *SQL Call Level Interface* (SQL/CLI) konnte sich dank der Umsetzung unter dem Namen Open Database Connectivity (ODBC) durch die Firma Microsoft und die parallel erfolgte internationale Normierung unter dem Titel *SQL/CLI* breit am Markt etablieren.

Die für die Programmiersprache Java adaptierte Variante des Zugriffs auf relationale DBMS wird durch SUN Microsystems unter dem Namen *Java Database Connectivity* (JDBC) propagiert und

stellt eine auf ODBC konzeptionell aufbauende und auf die spezifischen Bedürfnisse dieser Applikationsprogrammiersprache optimierte Untermenge des SQL/CLI-Standards dar.

Konzept und Grundidee

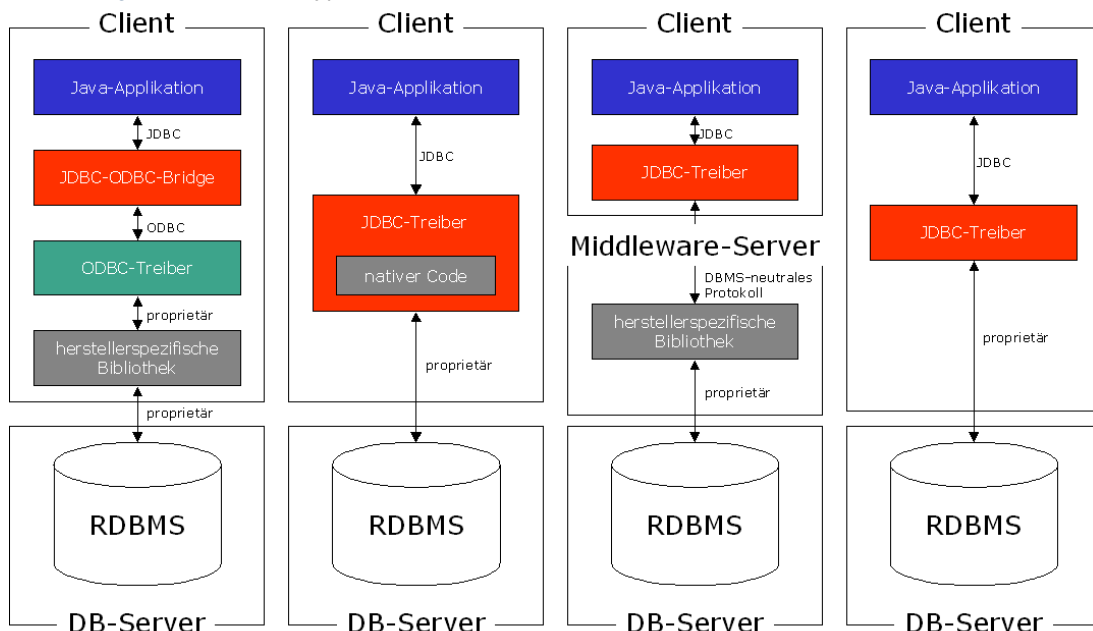
Von den Vorgängeransätzen übernommene Grundidee der Schnittstelle ist es den physischen Zugriff auf das Datenbankmanagementsystem durch eine von der Applikation spearierte wiederverwendbare Softwarekomponente, den sog. *JDBC-Treiber*, abzuwickeln.

Dieser Treiber vermittelt zwischen der Javaapplikation und dem verwendeten DBMS. Hierbei muß für jedes DBMS ein auf es abgestimmter JDBC-Treiber verwendet werden, da lediglich die Schnittstelle zur Applikation, nicht jedoch die zum DBMS, standardisiert ist.

Diesem Treiber obliegt die Abwicklung der gesamten Kommunikationsvorgänge mit dem DBMS. Er setzt jedoch selbst keine datenbankspezifischen Funktionalitäten, wie Syntax- oder Plausibilitätsprüfungen der übermittelten Kommandos um. Etwaige Fehlerprüfungen können, ebenso wie Anfrageoptimierungen, daher erst seitens des DBMS vorgenommen werden. Der Vorteil dieses Vorgehens liegt in der Generizität des JDBC-Treibers. Er kann ohne aufwendige Logikanteile als reine uninterpretierende Vermittlungsschicht zwischen Applikation und DBMS umgesetzt werden, wodurch schlanke Implementierungen ermöglicht werden.

Die [JDBC-Spezifikation](#) detailliert den Treiberbegriff zusätzlich hinsichtlich der gewählten technischen Umsetzung aus. So werden die vier in [Abbildung 1](#) dargestellten Treibertypen gemäß ihrer Charakteristika beschrieben und unterschieden.

Abbildung 1: JDBC-Treibertypen



Typ 1-Treiber

(click on image to enlarge!)

Typ 2-Treiber

Typ 3-Treiber

Typ 4-Treiber

Die historisch älteste Variante bildet der **Typ 1 Treiber**. Strenggenommen verkörpert er selbst keinen Datenbanktreiber, sondern lediglich eine Umsetzungsschicht die einem existierenden ODBC-Treiber vorgeschaltet wird.

Die Abbildung belegt diesen Treibertyp daher mit dem Begriff *JDBC-ODBC-Bridge*, da er lediglich den Brückenschlag zwischen den beiden Standards vornimmt und sich in der konkreten Anwendung auf die Umsetzung der zwischen den beiden Protokollen beschränkt, ohne realen Zugriff auf die Datenbank zu erhalten.

Dieser ist dem ODBC-Treiber vorbehalten, der im allgemeinen Falle mit einer weiteren Umsetzungsstufe kommuniziert, welche die generischen ODBC-Aufrufe in konkrete DBMS-spezifische wandelt.

Während sowohl der JDBC-ODBC-Brückentreiber als auch der ODBC-Treiber selbst für verschiedene DBMS verwendet werden können, muß für jedes konkrete DBMS eine herstellerspezifische, d.h. an das verwendete DBMS angepaßte, Bibliothek vorliegen.

Für den Fall eines **Typ 2 Treibers** entfällt diese durch ODBC geschaffene zusätzliche Indirektionsstufe zugunsten der Adaption der Konversionskomponente, welcher die Wandlung der Aufrufe in das DBMS-native Protokoll obliegt, an das JDBC-Protokoll und ihrer Integration in den JDBC-Treiber selbst.

Die Natur der Kommunikation des Java-Anteils des Treibers mit den Nativen ist im Rahmen der durch die JDBC-Spezifikation gegebenen Definition nicht festgelegt. Durch die Integration der DBMS-nativen Treiberanteile in den JDBC-Treiber muß dieser für jedes anzusprechende DBMS neu erstellt werden. Eine Wiederverwendung der JDBC-spezifischen Anteile die für die Clientkommunikation eingesetzt werden kann hierbei nicht erfolgen.

Der Fall der (partiellen) Konkretisierung dieser Kommunikationsbeziehung zu einem beliebigen *DBMS-neutralen Protokoll* wird durch einen **Typ 3 Treiber** aufgegriffen. Hier wird die DBMS-spezifische Komponente (in der Abbildung grau dargestellt) als vom JDBC-Treiber separiertes Modul aufgefaßt, daß mit diesem mittels eines festgelegten neutralen Protokolls kommuniziert.

Durch diese Separierung, die auch durch Installation auf physisch getrennten Maschinen --- der DBMS-spezifische Anteil könnte beispielsweise auf einem Middleware-Server untergebracht werden --- fundiert werden kann, gelingt die Wiederverwendung des JDBC-Treiberanteils, der mit verschiedenen DBMS-spezifischen Bibliotheken über das gewählte Protokoll kommunizieren kann.

Der **Typ 4 Treiber** stellt die letzte durch die JDBC-Spezifikation vorgesehene Ausprägung dar. Er konzipiert eine vollständig in Java implementierte Zugriffsschicht, die in sich geschlossen ist. Sie besitzt daher lediglich die notwendige JDBC-Schnittstelle zur Kommunikation mit der Java-Applikation und eine DBMS-Spezifische zum Zugriff auf die Datenquelle. Die Vorteile dieser Architekturvariante liegen in ihrer Portabilität und den geringen Installations und Wartungsaufwänden, die aus der Reduktion der Kommunikationsbeziehungen resultieren. So kann ein solcher Treiber durch einfache Integration in die Java-Applikation verwendet werden und bedarf keiner Installationen oder Modifikationen an der verwendeten Ausführungsumgebung. Gleichzeitig offenbart sich diese Lösung jedoch als technisch aufwendig in der Umsetzung, sobald DBMS verschiedener Hersteller angesprochen werden sollen, da die JDBC-Anteile des Treibers nicht separat wiederverwendet werden können.

Hinsichtlich des Laufzeitverhaltens zeigt sich deutlich die Schwäche der Typ 1 Treiber, welche in der inhärent notwendigen Doppelkonversion (JDBC zu ODBC und ODBC zu nativem Aufruf) begründet liegt. Daher sind Treiber dieses Typs als Übergangserscheinung hin zu „echten“ JDBC-Treibern, d.h. Treibern der restlichen Typen, anzusehen und sollten in Produktivumgebungen nicht eingesetzt werden.

Die Vorteile der Typ 2 und 3 Treiber seitens der Ausführungsgeschwindigkeit liegen in den nativen Codeanteilen begründet, welche für das jeweilige verwendete DBMS optimiert werden können. Zwar spricht der leichte Installations- und Administrationsaufwand eindeutig für Typ 4 Treiber, jedoch fallen diese in ihrer Leistungsfähigkeit durch die ausschließliche Verwendung der Programmiersprache Java teilweise deutlich hinter Treiber des Typs 2 und 3, mitunter sogar hinter solche des Typs 1, zurück. Sie verkörpern jedoch den aus konzeptioneller Sicht zu bevorzugenden Ansatz hinsichtlich Portabilität und Vergleichbarkeit der erzielten quantitativen Ergebnisse. Typischerweise kommen im produktiven Einsatz jedoch Treiber der Typen 2 und 4 zum Einsatz, die entweder durch den Hersteller des DBMS mitgeliefert werden (Typ 2) oder auf der Basis publizierter Schnittstellen plattformunabhängig für genau ein spezifisches DBMS entwickelt wurden (Typ 4).

Generell formuliert das JDBC-Konzept auf dieser Ebene noch keine Einschränkung hinsichtlich der unterstützten DBMS-Typen und ist generell auf verschiedenste Datenquellen anwendbar. Durch die Struktur des API und die verfügbaren Treiber kristallisieren sich jedoch relationale DBMS als Hauptanwendungsgebiet dieser Zugriffsschnittstelle heraus.

Im folgenden wird die Verwendung des Typ 4 Treibers [Connector/J](#) im Zusammenspiel mit dem RDBMS *MySQL* betrachtet.

Die Beispiele basieren auf einer Demodatenbank, deren Struktur und Inhalte nachfolgend angegeben sind.

Die Tabelle *EMPLOYEE*

```

+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+
| FNAME  | MINIT | LNAME  | SSN      | BDATE      | ADDRESS                                     |
SEX | SALARY  | SUPERSSN | DNO |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+
| John    | B     | Smith   | 123456789 | 1965-01-09 | 731 Fondren, Houston, TX |
M   | 30000.00 | 333445555 | 5 |
| Franklin | T     | Wong    | 333445555 | 1955-12-08 | 638 Voss, Houston, TX   |
M   | 40000.00 | 888665555 | 5 |
| Joyce   | A     | English | 453453453 | 1972-07-31 | 5631 Rice, Houston, TX  |
F   | 25000.00 | 333445555 | 5 |

```

M	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX
M	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX
F	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX
M	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX
F	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX

Umsetzung in der Java-API

Das Klassendiagramm der [Abbildung 2](#) zeigt die zentralen Klassen des Paketes `java.sql`. Auffallend ist, daß alle Elemente des dargestellten Pakets -- abgesehen von den definierten Exceptionklassen -- als Schnittstellen ausgelegt sind. Durch diese Mimik wird die Organisation der JDBC-Schnittstelle deutlich. Die API legt lediglich das Verhalten hinsichtlich seiner Semantik und die Einzeloperationen durch Definition ihrer Parameter fest, die konkrete DBMS-spezifische Implementierung dieser Operationen wird durch den JDBC-Treiber bereitgestellt.

Zentrale Klasse der JDBC-API ist die Schnittstelle [Connection](#). Sie bildet die Kommunikationsverbindungen zum DBMS ab und bietet notwendige Verwaltungsoperationen. Hierunter fallen insbesondere auch die Aufrufe zur Transaktionssteuerung.

Die Schnittstelle [Statement](#) realisiert genau eine aus Javasicht atomare Datenbankaktion. Diese muß hierbei aus minimal einem Aufruf an das DBMS bestehen, kann aber eine Reihe separater Aufrufe zu einem *Batch* bündeln.

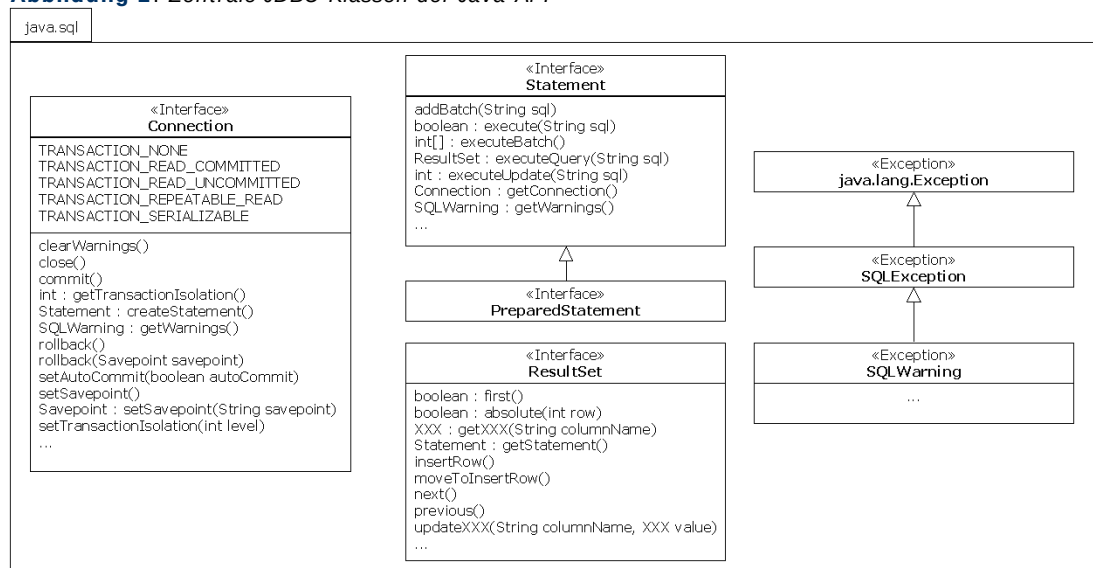
Als Sonderform sieht die API die Spezialisierung [PreparedStatement](#) vor, die es gestattet parametrisierte Anfragen zwischenspeichern, die nach Belegung der Parameterfelder an das DBMS übergeben werden. Hierdurch wird ein einfacher Mechanismus zur Wiederverwendung von DBMS-Aufrufen etabliert.

Liefert eine DBMS-Anfrage Ergebnistupel, wo werden diese konform zur Schnittstelle [ResultSet](#) verwaltet. Diese Schnittstelle erlaubt die lesende Traversierung der vom DBMS gelieferten Tupel ebenso wie ihre Aktualisierung im Hauptspeicher und das anschließende Zurückschreiben in die Datenbank.

Die in der Abbildung nur durch *getXXX* und *updateXXX* angedeuteten Operationen existieren in Ausprägungen für alle unterstützten Datentypen, wobei XXX den Namen des Typs bezeichnet.

Ferner definiert die API eine mit [SQLWarning](#) eine Ausnahme zur Behandlung auftretender Fehlersituationen sowie eine Reihe weiterer, in der [Abbildung 2](#) nicht dargestellter Klassen wie beispielsweise verschiedene Datentypen.

Abbildung 2: Zentrale JDBC-Klassen der Java-API



(click on image to enlarge!)

Zugriff auf die Datenbank

Beispiel 1 zeigt den Ablauf zur Aufnahme einer Verbindung mit der Datenbank jdbctest auf dem lokalen Rechner (localhost).

Zunächst muß die Klasse des gewählten JDBC-Treibers (im Beispiel `com.mysql.jdbc.Driver` vor ihrer Verwendung geladen werden. Dies geschieht durch den Aufruf der statischen Methode `forName` auf der Klasse `Class`.

Der zu ladende Treiber muß hierbei die JDBC-Schnittstellenklasse `Driver` implementieren um später durch die JDBC-API verwendet werden zu können.

Gleichzeitig mit dem dynamischen Ladevorgang erfolgt die Registrierung des Treibers beim JDBC-`DriverManager`, der die Verwaltung der geladenen DB-Treiber übernimmt.

Nach dem erfolgreichen Laden des Treibers wird durch den Aufruf von `getConnection` (Zeile 16) die Verbindung zur Datenbank hergestellt. Die anzusprechende Datenbank wird hierbei durch eine URI der Form `jdbc:mysql://DB-Server/DB-Name` repräsentiert (Zeile 17). Zusätzlich können ein zur Anmeldung am DB-System benötigter Benutzer (Zeile 18) und sein Paßwort (Zeile 19) übergeben werden.

Beispiel 1: Aufbau einer Datenbankverbindung



```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)
(5)public class JDBCConnect {
(6)    public static void main(String[] args) {
(7)        try {
(8)            Class.forName("com.mysql.jdbc.Driver");
(9)        } catch (ClassNotFoundException e) {
(10)            System.err.println("Driver class not found");
(11)            e.printStackTrace();
(12)        }
(13)        Connection con = null;
(14)        try {
(15)            con =
(16)                (Connection) DriverManager.getConnection(
(17)                    "jdbc:mysql://localhost/jdbctest/",
(18)                    "mario",
(19)                    "thePassword");
(20)        } catch (SQLException e1) {
(21)            System.err.println("Error establishing database connection");
(22)            e1.printStackTrace();
(23)        }
(24)    }
(25)
(26)}
```

[Download des Beispiels](#)

Zusätzlich stellen die Klassen `Driver` und `DriverManager` die Möglichkeit der Abfrage von verbindungsunabhängigen Verwaltungsinformationen zur Verfügung.

Beispiel 2: Ermittlung von Informationen über Treiber und Treibermanager

```
(1)import java.sql.Driver;
(2)import java.sql.DriverManager;
(3)import java.util.Enumeration;
(4)
(5)public class JDBCDriver {
(6)
(7)    public static void main(String[] args) {
(8)        try {
(9)            Class.forName("com.mysql.jdbc.Driver");
(10)        } catch (ClassNotFoundException e) {
(11)            System.err.println("Driver class not found");
(12)            e.printStackTrace();
(13)        }
(14)
(15)        System.out.println(
(16)            "DriverManager:\nlogin timeout=" + DriverManager.
getLoginTimeout());
(17)
(18)        Enumeration e = DriverManager.getDrivers();
```



```

(19)         while (e.hasMoreElements()) {
(20)             Driver drv = (Driver) e.nextElement();
(21)
(22)             System.out.println(
(23)                 "Driver="
(24)                     + drv.getClass().getName()
(25)                     + "\nmajor version="
(26)                     + drv.getMajorVersion()
(27)                     + "\nminor version="
(28)                     + drv.getMinorVersion()
(29)                     + "\nJDBC compliant="
(30)                     + drv.jdbcCompliant());
(31)         }
(32)
(33)     }
(34)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Beispiel 2 zeigt die Ermittlung des durch den DriverManager für alle durch ihn verwalteten Treiber global definierten Login Timouts, der angibt wie lange beim Anmeldevorgang an der Datenbank auf eine Rückmeldung gewartet wird.

Zusätzlich werden für alle verwalteten Treiber der Klassenname sowie Daten zur Version und zum Stand der JDBC-Unterstützung ermittelt und ausgegeben.

Der JDBC-Unterstützungsstand gibt an, ob ein gegebener Treiber die Konformitätstests der Firma SUN bestanden hat. Voraussetzung hierfür ist u.a. die vollständige Unterstützung des SQL 92-Standards (entry level).

Diese Interpretation von Spezifikationskonformität verwundert etwas, da alle JDBC-Treiber mit Ausnahme der inhärent DB-neutralen [Typ 1 Treiber](#) DBMS-spezifisch realisiert sind. Aus diesem Grunde bewertet der Konformitätstest vielmehr den Umsetzungsgrad des SQL-Standards in dem via JDBC genutzten DBMS als die Güte des JDBC-Treibers selbst.

Seit der JDBC-Schnittstellenversion 2 ist neben der „klassischen“ Zugriffsvariante auch eine auf dem [Java Naming and Directory Interface](#) (JNDI) basierende Zugriffsmethodik definiert, deren Verwendung --- abgesehen von der geänderten Mimik im Aufbau der DB-Verbindung --- identisch gestaltet ist.

Jedoch ist, wie in JNDI üblich, vor dem Zugriff ein benanntes Objekt beim JNDI-Dienst zu registrieren.

Im Falle von JDBC ist dies ein Objekt welches die Schnittstelle [DataSource](#) implementiert.

Der Code des Beispiels 3 zeigt die notwendigen Schritte zur Registrierung eines `MysqlDataSource`-Objekts, der durch den MySQL-JDBC-Treiber gelieferten Implementierung der Schnittstelle `DataSource`.

Beispiel 3: Ablage von Verbindungsinformation in einem JNDI-Verzeichnis

```

(1)import java.util.Hashtable;
(2)import javax.naming.Context;
(3)import javax.naming.InitialContext;
(4)import javax.naming.NamingException;
(5)import com.mysql.jdbc.jdbc2.optional.MysqlDataSource;
(6)
(7)public class JDBCConnect2Server {
(8)
(9)    public static void main(String[] args) {
(10)        Hashtable env = new Hashtable();
(11)        env.put(
(12)            Context.INITIAL_CONTEXT_FACTORY,
(13)            "com.sun.jndi.fscontext.ReffFSContextFactory");
(14)        env.put(Context.PROVIDER_URL, "file:/tmp/registry");
(15)
(16)        MysqlDataSource ds = new MysqlDataSource();
(17)        ds.setDatabaseName("jdbctest");
(18)        Context ctx = null;
(19)        try {
(20)            ctx = new InitialContext(env);
(21)        } catch (NamingException ne) {
(22)            ne.printStackTrace();

```



```

(23)          }
(24)
(25)          try {
(26)              ctx.rebind("jdbc/mySrc", ds);
(27)          } catch (NamingException ne) {
(28)              ne.printStackTrace();
(29)          }
(30)      }
(31)}

```

[Download des Beispiels](#)

Entsprechend der modifizierten Ablage der Verwaltungsinformation ändert sich die Erzeugung der Datenbankverbindung beim Zugriff. Hier wird nun zunächst über einen Zugriff auf den JNDI-Verzeichnisdienst das benannte DataSource-Objekt (es trägt den Namen jdbc/mySrc ermittelt. Anschließend wird durch das dem Verzeichnisdienst entnommene DataSource-Objekt die Datenbankverbindung (d.h. das Connection-Objekt) erzeugt.

Alle weiteren Schritte zur Interaktion mit der Datenbank verlaufen dann identisch zur im Beispiel 1 gezeigten Verbindungsaufnahme.

Der Code des Beispiels 4 zeigt die notwendigen Schritte zur Ermittlung der Referenz auf das Objekt des Typs DataSource aus dem JNDI-Verzeichnis, sowie die Erzeugung des Connection-Objekts.

Beispiel 4: Verbindungsaufbau unter Nutzung von JNDI

```

(1)import java.sql.Connection;
(2)import java.sql.SQLException;
(3)import java.util.Hashtable;
(4)import javax.naming.Context;
(5)import javax.naming.InitialContext;
(6)import javax.naming.NamingException;
(7)import javax.sql.DataSource;
(8)
(9)public class JDBCConnect2 {
(10)    public static void main(String[] args) {
(11)        Hashtable env = new Hashtable();
(12)        env.put(
(13)            Context.INITIAL_CONTEXT_FACTORY,
(14)            "com.sun.jndi.fscontext.RefFSContextFactory");
(15)        env.put(Context.PROVIDER_URL, "file:/tmp/registry");
(16)        Context ctx = null;
(17)        try {
(18)            ctx = new InitialContext(env);
(19)        } catch (NamingException ne) {
(20)            ne.printStackTrace();
(21)        }
(22)        DataSource ds = null;
(23)        try {
(24)            ds = (DataSource) ctx.lookup("jdbc/mySrc");
(25)        } catch (NamingException ne) {
(26)            ne.printStackTrace();
(27)        }
(28)        Connection con = null;
(29)        try {
(30)            con = ds.getConnection("mario", "thePassword");
(31)        } catch (SQLException sqle) {
(32)            sqle.printStackTrace();
(33)        }
(34)    }
(35)}

```



[Download des Beispiels](#)

Auffallend ist die Ablage des Datenbanknamens im Verzeichnisdienst mittels des Methodenaufrufs setDatabaseName. Diese Verschiebung der Information wird durch die geänderte Mimik der Erzeugung des Connection-Objekts impliziert. So sieht die Implementierung dieser Methode für die Klasse DataSource keine Möglichkeit zur gleichzeitigen Übergabe von Anmeldenamen, Paßwort und Datenbank vor.

Vielmehrnoch ist es sogar möglich diese Daten allesamt innerhalb des JNDI-Verzeichnisdienstes abzulegen. (Für diesen Zweck stehen die Methoden setUser bzw. setPassword zur Verfügung.) Als Konsequenz hiervon kann der Verbindungswunsch durch Aufruf der Methode getConnection ohne

weitere Parameter erfüllt werden.

Diese Umsetzungsweise ist vor ihrer Realisierung hinsichtlich des damit eintretenden Verlustes an Sicherheit zu prüfen, da in ihrer Folge eine Datenbankverbindung allein durch Kenntnis des JNDI-residenten Namens des DataSource-Objektes erfolgen kann.

Generell wählen JDBC-Umsetzungen den Weg jede Ausprägung eines Connection-Objekts in eine physische Datenbankverbindung abzubilden. Dieses, durchaus der intuitiven Semantik der Connection-Klasse entsprechende Vorgehen kann jedoch in realen Applikationen, begründet in der Vielzahl der durch das DBMS zu verwaltenden Verbindungen, zu Zugriffsengpässen führen.

Aus diesem Grunde definiert die JDBC-Schnittstelle Operationen zur Zusammenfassung „gleichartiger“ Zugriffe. Hierzu zählen Zugriffe die unter derselben Nutzerkennung auf dieselbe Datenbank abgewickelt werden. Diese Zugriffsform tritt insbesondere bei Anwendungen auf, die über nur einen in der Datenbank eingetragenen Anwender verfügen und die gesamte Nutzerverwaltung datenbanktransparent applikationsseitig abwickeln.

Zur Optimierung von Zugriffen dieser Natur sieht die JDBC-Schnittstelle das sog. *Connection Pooling* vor, welches gleichartige Zugriffe bündelt.

Das Beispiel 5 zeigt eine Umsetzung:

Beispiel 5: Verbindungsaufbau unter Nutzung von Connection Pooling



```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import javax.sql.PooledConnection;
(4)import com.mysql.jdbc.Connection;
(5)import com.mysql.jdbc.jdbc2.optional.MysqlPooledConnection;
(6)
(7)public class JDBCConnection3 {
(8)    public static void main(String[] args) {
(9)        try {
(10)            Class.forName("com.mysql.jdbc.Driver");
(11)        } catch (ClassNotFoundException cnfe) {
(12)            System.err.println("Driver class not found");
(13)            cnfe.printStackTrace();
(14)        }
(15)        Connection con = null;
(16)        try {
(17)            con =
(18)                (Connection) DriverManager.getConnection(
(19)                    "jdbc:mysql://localhost/jdbctest/",
(20)                    "mario",
(21)                    "thePassword");
(22)        } catch (SQLException e1) {
(23)            System.err.println("Error establishing database connection");
(24)            e1.printStackTrace();
(25)        }
(26)
(27)        PooledConnection pc = new MysqlPooledConnection(con);
(28)
(29)        java.sql.Connection con1 = null;
(30)        try {
(31)            con1 = pc.getConnection();
(32)        } catch (SQLException sqle) {
(33)            sqle.printStackTrace();
(34)        }
(35)    }
(36)}
```

[Download des Beispiels](#)

Statt für jede gewünschte Datenbankverbindung ein zusätzliches Objekt des Type Connection zu erzeugen wird die erzeugte Verbindung zur Konstruktion eines Objektes, welches Konform zur Schnittstelle [PooledConnection](#) definiert ist, verwendet. Dieses verwaltet sort für die Verwaltung der DB-Verbindung und stellt dieselbe physische Verbindung verschiedenen Anfragern zur Verfügung.

Konsequenterweise wird daher eine neue Verbindung nicht mehr vom DriverManager angefordert, sondern durch die Methode getConnection der aus der Verwaltungsstruktur entnommenen PooledConnection beantragt.

Aufgrund der Unterstützung des SQL-Sprachumfanges, durch unveränderte textuelle Propagation an das DBMS sind durch JDBC im Allgemeinen alle Facetten der Datenbanksprache nutzbar, sofern

sie durch das verwendete DBMS Unterstützung finden. Hierunter fallen:

- Data Definition Language.
Zur Erzeugung eines Datenmodells.
- Data Manipulation Language.
Zur Modifikation der verwalteten Daten.
- Data Retrieval Language.
Zur Anfrage der in einer Datenbank gespeicherten Daten.
- Data Control Language.
Zur Festlegung und Kontrolle von Zugriffsberechtigungen.

JDBC reflektiert jedoch nicht diese Sprach(-sub-)klassen selbst in der API, sondern sieht vielmehr ausschließlich zwei Formen des Zugriffs vor. Solche die tabellenwerte Resultate liefern und solche, deren Ausführung lediglich primitivwertige Rückgabewerte liefert.

Primitivwertige Zugriffe

Primitivwertige Datenbankzugriffe liefern, abgesehen von Fehler- oder Warnmeldungen, lediglich die Anzahl der geänderten Tupel, falls zutreffend, oder 0 zurück.

Aus dieser Festlegung lassen sich diejenigen SQL-Anweisungstypen ableiten, welche als primitivwertiger Zugriff realisiert sind. Hierunter fallen alle Operationen der Datendefinition wie CREATE oder ALTER TABLE sowie alle Einfüge- (INSERT) Änderungs- (UPDATE) und Löschvorgänge (DELETE). Darüberhinaus alle Operationen zur Administration der Datenbank durch Rechtevergabe (GRANT, REVOKE).

Zugriffe dieser Art werden generell durch die Methode [executeUpdate](#), oder einer Abart davon, realisiert.

Beispiel 6: Erstellung einer neuen Tabelle

```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCCreateTable {
(7)    public static void main(String[] args) {
(8)        try {
(9)            Class.forName("com.mysql.jdbc.Driver");
(10)        } catch (ClassNotFoundException e) {
(11)            System.err.println("Driver class not found");
(12)            e.printStackTrace();
(13)        }
(14)        Connection con = null;
(15)
(16)        try {
(17)            con =
(18)                (Connection) DriverManager.getConnection(
(19)                    "jdbc:mysql://localhost/jdbctest/",
(20)                    "mario",
(21)                    "thePassword");
(22)        } catch (SQLException e1) {
(23)            System.err.println("Error establishing database connection");
(24)            e1.printStackTrace();
(25)        }
(26)
(27)        Statement stmt = null;
(28)        try {
(29)            stmt = (Statement) con.createStatement();
(30)        } catch (SQLException e2) {
(31)            System.err.println("Error creating SQL-Statement");
(32)            e2.printStackTrace();
(33)        }
(34)        String createTab = new String("CREATE TABLE EMPLOYEE(" +
(35)            "FNAME VARCHAR(10) NOT NULL," +
(36)            "MINIT VARCHAR(1)," +
(37)            "LNAME VARCHAR(10) NOT NULL," +
(38)            "SSN INTEGER(9) NOT NULL," +
(39)            "BDATE DATE," +
(40)            "ADDRESS VARCHAR(30)," +
(41)            "SEX ENUM('M','F')," +
(42)            "SALARY REAL(7,2) UNSIGNED," +
```



```

(43)         "SUPERSN INTEGER(9)," +
(44)         "DNO INTEGER(1));";
(45)     try {
(46)         System.out.println("result="+stmt.executeUpdate(createTab));
(47)     } catch (SQLException e3) {
(48)         System.err.println("Error creating table EMPLOYEE");
(49)         e3.printStackTrace();
(50)     }
(51) }
(52)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Beispiel 6 zeigt die notwendigen Schritte zur Erstellung der [Tabelle EMPLOYEE](#) in der Datenbank.

Nach dem (üblichen) Verbindungsaufbau (Zeile 8-24) wird in Zeile 26 eine Variable des Typs [Statement](#) deklariert. Auch bei Statement handelt es sich um eine durch die JDBC-API vordefinierte Schnittstelle die als Bestandteil des JDBC-Treibers durch von einer Klasse implementiert wird.

Ausgehend von der etablierten Datenbankverbindung wird durch Aufruf der Methode [createStatement](#) eine konkrete Ausprägung konform zur Statement-Schnittstelle erzeugt (Zeile 28).

Der Aufruf von [executeUpdate](#) übergibt das als Zeichenkette abgelegte SQL-Kommando an die Datenbank zur Ausführung.

Da durch CREATE TABLE keine Tupeländerungen vorgenommen werden ist das Resultat des Aufrufs der Rückgabewert 0.

Beispiel 7 zeigt mit dem ALTER TABLE-Kommando eine weitere Anwendung der executeUpdate-Methode.

Auch in diesem Falle wird als Resultat 0 geliefert, da die Definition des Primärschlüssels keine Änderungen an den verwalteten Datensätzen vornimmt.

Beispiel 7: Modifikation der Tabellendefinition

```

(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCAlterTable {
(7)    public static void main(String[] args) {
(8)        try {
(9)            Class.forName("com.mysql.jdbc.Driver");
(10)        } catch (ClassNotFoundException e) {
(11)            System.err.println("Driver class not found");
(12)            e.printStackTrace();
(13)        }
(14)        Connection con = null;
(15)
(16)        try {
(17)            con =
(18)                (Connection) DriverManager.getConnection(
(19)                    "jdbc:mysql://localhost/jdbctest/",
(20)                    "mario",
(21)                    "thePassword");
(22)        } catch (SQLException e1) {
(23)            System.err.println("Error establishing database connection");
(24)            e1.printStackTrace();
(25)        }
(26)
(27)        Statement stmt = null;
(28)        try {
(29)            stmt = (Statement) con.createStatement();
(30)        } catch (SQLException e2) {
(31)            System.err.println("Error creating SQL-Statement");
(32)            e2.printStackTrace();
(33)        }
(34)        String createTab =
(35)            new String("ALTER TABLE EMPLOYEE ADD PRIMARY KEY (SSN);");

```



```

(36)          try {
(37)              System.out.println("result=" + stmt.executeUpdate
(createTab));
(38)          } catch (SQLException e3) {
(39)              System.err.println("Error altering table EMPLOYEE");
(40)              e3.printStackTrace();
(41)          }
(42)      }
(43)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Beispiel 8: Einfügen von Werten

```

(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCInsert1 {
(7)    public static void main(String[] args) {
(8)        try {
(9)            Class.forName("com.mysql.jdbc.Driver");
(10)        } catch (ClassNotFoundException e) {
(11)            System.err.println("Driver class not found");
(12)            e.printStackTrace();
(13)        }
(14)        Connection con = null;
(15)
(16)        try {
(17)            con =
(18)                (Connection) DriverManager.getConnection(
(19)                    "jdbc:mysql://localhost/jdbctest/",
(20)                    "mario",
(21)                    "thePassword");
(22)        } catch (SQLException e1) {
(23)            System.err.println("Error establishing database connection");
(24)            e1.printStackTrace();
(25)        }
(26)
(27)        Statement stmt = null;
(28)        try {
(29)            stmt = (Statement) con.createStatement();
(30)        } catch (SQLException e2) {
(31)            System.err.println("Error creating SQL-Statement");
(32)            e2.printStackTrace();
(33)        }
(34)
(35)        try {
(36)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('John', 'B', 'Smith', 123456789, '1965-01-09', '731 Fondren,
Houston, TX', 'M', 30000, 333445555, 5);"));
(37)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Franklin', 'T', 'Wong', 333445555, '1955-12-08', '638 Voss,
Houston, TX', 'M', 40000, 888665555, 5);"));
(38)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Alicia', 'J', 'Zelaya', 999887777, '1968-07-19', '3321 Castle,
Spring, TX', 'F', 25000, 987654321, 4);"));
(39)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Jennifer', 'S', 'Wallace', 987654321, '1941-06-20', '291
Berry, Bellaire, TX', 'F', 43000, 888665555, 4);"));
(40)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Ramesh', 'K', 'Narayan', 666884444, '1962-09-15', '975 Fire
Oak, Humble, TX', 'M', 38000, 333445555, 5);"));
(41)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Joyce', 'A', 'English', 453453453, '1972-07-31', '5631 Rice,
Houston, TX', 'F', 25000, 333445555, 5);"));
(42)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Ahmad', 'V', 'Jabbar', 987987987, '1969-03-29', '980 Dallas,
Houston, TX', 'M', 25000, 987654321, 4);"));

```



```

(43)                System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('James', 'E', 'Borg', 888665555, '1937-11-10', '450 Stone,
Houston, TX', 'M', 55000, null, 1);"));
(44)                } catch (SQLException e3) {
(45)                System.err.println("Error inserting values into table
EMPLOYEE");
(46)                e3.printStackTrace();
(47)                }
(48)        }
(49)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Beispiel 8 zeigt den Einfügevorgang von acht Werten in die durch die vorangegangenen Beispiele erzeugte Tabelle EMPLOYEE.

Jeder der Einfügevorgänge der Zeilen 36-43 führt im Rahmen einer separaten Datenbankkommunikation sequentiell genau einen Einfügevorgang durch, was durch den Rückgabewert 1 dokumentiert wird.

Zwar ist dieses Verfahren praktikabel und erzielt die angestrebten Resultate, jedoch ist es unter Zeiteffizienzgesichtspunkten inadäquat, da sich Einfüge- und Kommunikationsvorgänge zahlenmäßig entsprechen.

Aus diesem Grunde bietet die Schnittstelle [Statement](#) die Möglichkeit zur Bündelung einzelner SQL-Aufrufe in einem sog. *Batch* an.

Beispiel 9 zeigt die entsprechende Umgestaltung des vorangegangenen Beispiels.

Beispiel 9: Einfügen von Werten mittels eines Batches

```

(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCInsert2 {
(7)    public static void main(String[] args) {
(8)        try {
(9)            Class.forName("com.mysql.jdbc.Driver");
(10)       } catch (ClassNotFoundException e) {
(11)            System.err.println("Driver class not found");
(12)            e.printStackTrace();
(13)       }
(14)       Connection con = null;
(15)
(16)       try {
(17)           con =
(18)               (Connection) DriverManager.getConnection(
(19)                   "jdbc:mysql://localhost/jdbctest/",
(20)                   "mario",
(21)                   "thePassword");
(22)       } catch (SQLException e1) {
(23)            System.err.println("Error establishing database connection");
(24)            e1.printStackTrace();
(25)       }
(26)
(27)       Statement stmt = null;
(28)       try {
(29)           stmt = (Statement) con.createStatement();
(30)       } catch (SQLException e2) {
(31)            System.err.println("Error creating SQL-Statement");
(32)            e2.printStackTrace();
(33)       }
(34)
(35)       try {
(36)           stmt.addBatch("INSERT INTO EMPLOYEE VALUES('John', 'B',
'Smith', 123456789, '1965-01-09', '731 Fondren, Houston, TX', 'M', 30000, 333445555,
5);");
(37)           stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Franklin', 'T',
'Wong', 333445555, '1955-12-08', '638 Voss, Houston, TX', 'M', 40000, 888665555,

```



```

5);");
(38)                stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Alicia', 'J',
'Zelaya', 999887777, '1968-07-19', '3321 Castle, Spring, TX', 'F', 25000, 987654321,
4);");
(39)                stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Jennifer', 'S',
'Wallace', 987654321, '1941-06-20', '291 Berry, Bellaire, TX', 'F', 43000,
888665555, 4);");
(40)                stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Ramesh', 'K',
'Narayan', 666884444, '1962-09-15', '975 Fire Oak, Humble, TX', 'M', 38000,
333445555, 5);");
(41)                stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Joyce', 'A',
'English', 453453453, '1972-07-31', '5631 Rice, Houston, TX', 'F', 25000, 333445555,
5);");
(42)                stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Ahmad', 'V',
'Jabbar', 987987987, '1969-03-29', '980 Dallas, Houston, TX', 'M', 25000, 987654321,
4);");
(43)                stmt.addBatch("INSERT INTO EMPLOYEE VALUES('James', 'E',
'Borg', 888665555, '1937-11-10', '450 Stone, Houston, TX', 'M', 55000, null, 1);");
(44)                int[] insertCounts = stmt.executeBatch();
(45)                } catch (SQLException e3) {
(46)                System.err.println("Error inserting values into table
EMPLOYEE");
(47)                e3.printStackTrace();
(48)                }
(49)        }
(50)}

```

[Download des Beispiels](#)

Statt der Einzelübergabe der SQL INSERT-Anweisungen werden diese nun (in Zeile 36-43) in in einem Batch gesammelt. Hierzu werden die SQL-Zeichenketten durch den Aufruf [addBatch](#) innerhalb des Statement-Objekts abgelegt und durch Aufruf der Methode [executeBatch](#) gesammelt an das DBMS übergeben.

Statt der Einzelresultate wird durch diese Aufrufvariante ein Array geliefert, der die Einzelergebnisse der als Batch übergebenen Aufrufe versammelt.

Dies verdeutlicht nochmals das nachfolgende Beispiel. In ihm wird zunächst mittels ALTER TABLE eine neue Tabellenspalte zur Aufnahme des Wochentages der Geburt erstellt und anschließend durch SQL UPDATE-Anweisungen die benötigten Daten aus dem vorhandenen Geburtsdatum ermittelt.

Auch dieses Beispiel bedient sich zur Performancebeschleunigung der Möglichkeiten des Batchaufrufes.

Beispiel 10: Aktualisieren von Tabellendefinitionen und Werten

```

(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCUpdate1 {
(7)    public static void main(String[] args) {
(8)        try {
(9)            Class.forName("com.mysql.jdbc.Driver");
(10)        } catch (ClassNotFoundException e) {
(11)            System.err.println("Driver class not found");
(12)            e.printStackTrace();
(13)        }
(14)        Connection con = null;
(15)
(16)        try {
(17)            con =
(18)                (Connection) DriverManager.getConnection(
(19)                    "jdbc:mysql://localhost/jdbctest/",
(20)                    "mario",
(21)                    "thePassword");
(22)        } catch (SQLException e1) {
(23)            System.err.println("Error establishing database connection");
(24)            e1.printStackTrace();
(25)        }
(26)

```



```

(27)      Statement stmt = null;
(28)      try {
(29)          stmt = (Statement) con.createStatement();
(30)      } catch (SQLException e2) {
(31)          System.err.println("Error creating SQL-Statement");
(32)          e2.printStackTrace();
(33)      }
(34)
(35)      try {
(36)          stmt.addBatch("ALTER TABLE EMPLOYEE ADD BDAY VARCHAR(10);");
(37)          stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Sunday' WHERE
DAYOFWEEK(BDATE)=1;");
(38)          stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Monday' WHERE
DAYOFWEEK(BDATE)=2;");
(39)          stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Tuesday' WHERE
DAYOFWEEK(BDATE)=3;");
(40)          stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Wednesday' WHERE
DAYOFWEEK(BDATE)=4;");
(41)          stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Thursday' WHERE
DAYOFWEEK(BDATE)=5;");
(42)          stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Friday' WHERE
DAYOFWEEK(BDATE)=6;");
(43)          stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Saturday' WHERE
DAYOFWEEK(BDATE)=7;");
(44)          int[] result = stmt.executeBatch();
(45)          for (int i=0; i<result.length;i++){
(46)              System.out.println("Statement No "+i+" changed
"+result[i]+" rows");
(47)          }
(48)      } catch (SQLException e3) {
(49)          System.err.println("Error inserting values into table
EMPLOYEE");
(50)          e3.printStackTrace();
(51)      }
(52)  }
(53)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Die Ausführung liefert als Resultat:

```

Statement No 0 changed 8 rows
Statement No 1 changed 0 rows
Statement No 2 changed 1 rows
Statement No 3 changed 0 rows
Statement No 4 changed 1 rows
Statement No 5 changed 1 rows
Statement No 6 changed 2 rows
Statement No 7 changed 3 rows

```

So werden durch den ALTER TABLE-Aufruf (Indexnummer 0) alle acht Tupel der Tabelle modifiziert, während die nachfolgenden Aufrufe nur Teilmengen davon verändern.

Die nähere Betrachtung der Zeilen 37-43 des Quellcodes von Beispiel 10 zeigt sich, daß diese im Kern denselben Vorgang ausführen, nur jeweils mit variierenden Parametern.

Zur Behandlung von Fällen dieser Problemstellung definiert die JDBC-API die Schnittstelle [PreparedStatement](#) als Spezialisierung von Statement.

Diese Schnittstelle gestattet es Anweisungen die später an die Datenbank übermittelt werden sollen mit Platzhaltern zu versehen und diese vor der Übermittlung mit Werten zu befüllen.

Beispiel 11 zeigt die entsprechende Modifikation des vorangegangenen Beispiels.

Beispiel 11: Aktualisieren von Tabellendefinitionen und Werten



```

(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.PreparedStatement;
(5)import com.mysql.jdbc.Statement;
(6)
(7)public class JDBCUpdate2 {
(8)    public static void main(String[] args) {
(9)        try {
(10)            Class.forName("com.mysql.jdbc.Driver");
(11)        } catch (ClassNotFoundException e) {
(12)            System.err.println("Driver class not found");
(13)            e.printStackTrace();
(14)        }
(15)        Connection con = null;
(16)
(17)        try {
(18)            con =
(19)                (Connection) DriverManager.getConnection(
(20)                    "jdbc:mysql://localhost/jdbctest/",
(21)                    "mario",
(22)                    "thePassword");
(23)        } catch (SQLException e1) {
(24)            System.err.println("Error establishing database connection");
(25)            e1.printStackTrace();
(26)        }
(27)
(28)        Statement stmt = null;
(29)        PreparedStatement pstmt = null;
(30)        try {
(31)            stmt = (Statement) con.createStatement();
(32)            pstmt = (PreparedStatement) con.prepareStatement("UPDATE
EMPLOYEE SET BDAY=? WHERE DAYOFWEEK(BDATE)=?");
(33)
(34)        } catch (SQLException e2) {
(35)            System.err.println("Error creating SQL-Statement");
(36)            e2.printStackTrace();
(37)        }
(38)
(39)        try {
(40)            String[] days=
{"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"} ;
(41)            stmt.addBatch("ALTER TABLE EMPLOYEE ADD BDAY VARCHAR(10);");
(42)            for (int i=1; i<8;i++){
(43)                pstmt.setString(1,days[i-1]);
(44)                pstmt.setInt(2,i);
(45)                pstmt.addBatch();
(46)            }
(47)            int[] result = stmt.executeBatch();
(48)            for (int i=0; i<result.length;i++){
(49)                System.out.println("Statement No "+i+" changed
"+"result[i]+" rows");
(50)            }
(51)        } catch (SQLException e3) {
(52)            System.err.println("Error inserting values into table
EMPLOYEE");
(53)            e3.printStackTrace();
(54)        }
(55)    }
(56)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Im Beispiel wird neben dem Objekt des Typs Statement zusätzlich eines des Typs PreparedStatement erzeugt (Zeile 32).

Die dem Konstruktor übergebene Anweisung enthält als Sonderzeichen zur Markierung der Platzhalter das Fragezeichen (?).

Die Wochentage werde in Zeile 40, des vereinfachten Zugriffs wegen, als Array definiert.

In den Zeilen 42 mit 46 werden die benötigten SQL-UPDATE-Anweisungen dynamisch aus durch

Einsetzen der geeigneten Werte in den vorpräparierten Änderungsausruck erzeugt und einem eigenen Batch zugeordnet. Der Einsetzungsvorgang der benötigten Werte geschieht durch die Methoden [setString](#) für zeichenkettenartige bzw. [setInt](#) für den ganzzahlige Parameter. Den Methoden wird jeweils die Position des Parameters, gezählt ab 1 sowie die zu wählende Wertbelegung übermittelt.

Zur Ausführung müssen beide Batches getrennt angefordert werden.

Tabellenwertige Zugriffe

Die in der Praxis quantitativ bedeutendste Klasse von Datenbankzugriffen dürfte zweifellos auf die lesende Ermittlung von bestehenden Daten darstellen, kurzum alle Spielarten der SQL SELECT-Anweisung.

Für Anfragen an die Datenbank steht prinzipiell der gesamte durch das DBMS unterstützte SQL-Umfang zur Verfügung.

Anfragen werden im Gegensatz zu den bisher betrachteten lesenden Zugriffen nicht als primivwerte Methoden realisiert, sondern liefern als Resultat immer eine Tabelle zurück.

Diese wird durch den API-Typ [ResultSet](#) dargestellt.

Zusätzlich werden Anfragen durch die Methode [executeQuery](#) ausgeführt.

Das Beispiel 12 zeigt die generische Extraktion von DB-Daten und den Zugriff auf Metadaten. Die aus der Datenbank gelesenen Ergebnistupel werden im durch rs benannten ResultSet abgelegt (Zeile 39). Die Resultatmenge wird mithilfe eines *Cursors* (Datensatzzeiger) traversiert. Hierzu wird der initial auf eine Ausgangsstellung vor dem ersten empfangenen Tupel positionierte Cursor durch Aufruf der Methode `next` solange weitergerückt, bis der letzte Datensatz verarbeitet wurde.

Der Aufruf der Methode [getMetaData](#) liefert deskriptive Metadaten wie Spaltenzahl sowie deren Bezeichner und Typen für die erstellte Resultattupelmenge.

In Zeile 43 werden diese Metadaten verwendet um die Spaltennamen der extrahierten Attribute anzuzeigen.

Zeile 47-52 liest die einzelnen Werte jedes Tupels mittels [getObject](#) aus und stellt sie am Bildschirm dar.

Beispiel 12: Auslesen von Daten und Metadaten

```
(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.ResultSetMetaData;
(4)import java.sql.SQLException;
(5)
(6)import com.mysql.jdbc.Connection;
(7)import com.mysql.jdbc.Statement;
(8)
(9)public class JDBCSelect1 {
(10)    public static void main(String[] args) {
(11)        try {
(12)            Class.forName("com.mysql.jdbc.Driver");
(13)        } catch (ClassNotFoundException e) {
(14)            System.err.println("Driver class not found");
(15)            e.printStackTrace();
(16)        }
(17)        Connection con = null;
(18)
(19)        try {
(20)            con =
(21)                (Connection) DriverManager.getConnection(
(22)                    "jdbc:mysql://localhost/jdbctest/",
(23)                    "mario",
(24)                    "thePassword");
(25)        } catch (SQLException e1) {
(26)            System.err.println("Error establishing database connection");
(27)            e1.printStackTrace();
(28)        }
(29)
(30)        Statement stmt = null;
(31)        try {
(32)            stmt = (Statement) con.createStatement();
(33)        } catch (SQLException e2) {
(34)            System.err.println("Error creating SQL-Statement");
```



```

(35)                e2.printStackTrace();
(36)            }
(37)
(38)            try {
(39)                ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(40)                ResultSetMetaData rsmd = rs.getMetaData();
(41)                int noColumns = rsmd.getColumnCount();
(42)                for (int i = 1; i < noColumns; i++) {
(43)                    System.out.print(rsmd.getColumnLabel(i) + "\t");
(44)                }
(45)                System.out.println();
(46)
(47)                while (rs.isLast() == false) {
(48)                    rs.next();
(49)                    for (int i = 1; i < noColumns; i++) {
(50)                        System.out.print( rs.getObject(i)+"\t" );
(51)                    }
(52)                    System.out.println();
(53)                }
(54)
(55)            } catch (SQLException e3) {
(56)                System.err.println("Error selecting values from table
EMPLOYEE");
(57)                e3.printStackTrace();
(58)            }
(59)        }
(60)    }

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Neben im Beispiel 12 gezeigten Verarbeitung in exakter der Ablagereihenfolge der Datenbank kann auch durch Definition eines Cursors die Traversierung in inverser Ablagerichtung erreicht werden. Das nachfolgende Beispiel das entsprechende Vorgehen durch anfängliche Positionierung des Cursors ans Ende der empfangenen Daten (d.h. nach dem letzten Datensatz) und anschließendes schrittweises Rückpositionieren durch Aufruf der Methode `previous`.

Beispiel 13: Auslesen von Daten in invertierter Reihenfolge

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import com.mysql.jdbc.Connection;
(5)import com.mysql.jdbc.Statement;
(6)
(7)public class JDBCSelect5 {
(8)    public static void main(String[] args) {
(9)        try {
(10)            Class.forName("com.mysql.jdbc.Driver");
(11)        } catch (ClassNotFoundException e) {
(12)            System.err.println("Driver class not found");
(13)            e.printStackTrace();
(14)        }
(15)        Connection con = null;
(16)
(17)        try {
(18)            con =
(19)                (Connection) DriverManager.getConnection(
(20)                    "jdbc:mysql://localhost/jdbctest/",
(21)                    "mario",
(22)                    "thePassword");
(23)        } catch (SQLException sqle) {
(24)            System.err.println("Error establishing database connection");
(25)            sqle.printStackTrace();
(26)        }
(27)
(28)        Statement stmt = null;
(29)        try {
(30)            stmt =
(31)                (Statement) con.createStatement();
(32)        } catch (SQLException e2) {
(33)            System.err.println("Error creating SQL-Statement");

```



```

(34)                e2.printStackTrace();
(35)            }
(36)
(37)            try {
(38)                ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(39)                rs.afterLast();
(40)                while (rs.previous()){
(41)                    System.out.println(rs.getString("FNAME"));
(42)                }
(43)            } catch (SQLException sqle) {
(44)                System.err.println("Error selecting values from table
EMPLOYEE");
(45)                sqle.printStackTrace();
(46)            }
(47)        }
(48)    }

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Ferner kann der Cursor wahlfrei auf eine beliebige Position der Ergebnisrelation gesetzt werden. Das nachfolgende Beispiel zeigt dies. Ferner illustriert es das Vorgehen zur Größenermittlung des resultierenden ResultSets durch das Aufrufpaar `last` und `getRow`, welches zunächst den Cursor auf den letzten aus der Datenbank extrahierten Datensatz positioniert und anschließend dessen Nummer liefert.

Beispiel 14: Auslesen von Daten in wahlfreier Reihenfolge

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import com.mysql.jdbc.Connection;
(5)import com.mysql.jdbc.Statement;
(6)
(7)public class JDBCSelect6 {
(8)    public static void main(String[] args) {
(9)        try {
(10)            Class.forName("com.mysql.jdbc.Driver");
(11)        } catch (ClassNotFoundException e) {
(12)            System.err.println("Driver class not found");
(13)            e.printStackTrace();
(14)        }
(15)        Connection con = null;
(16)
(17)        try {
(18)            con =
(19)                (Connection) DriverManager.getConnection(
(20)                    "jdbc:mysql://localhost/jdbctest/",
(21)                    "mario",
(22)                    "thePassword");
(23)        } catch (SQLException sqle) {
(24)            System.err.println("Error establishing database connection");
(25)            sqle.printStackTrace();
(26)        }
(27)
(28)        Statement stmt = null;
(29)        try {
(30)            stmt = (Statement) con.createStatement();
(31)        } catch (SQLException e2) {
(32)            System.err.println("Error creating SQL-Statement");
(33)            e2.printStackTrace();
(34)        }
(35)
(36)        try {
(37)            int position = 0;
(38)            ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(39)            rs.last();
(40)            int size = rs.getRow();
(41)            for (int i = 0; i < size; i++) {
(42)                position = (position + 3) % size;
(43)                rs.absolute(position + 1);

```



```

(44)                System.out.println(
(45)                "position=" + (position + 1) + ": " + rs.
getString("FNAME"));
(46)                }
(47)            } catch (SQLException sqle) {
(48)                System.err.println("Error selecting values from table
EMPLOYEE");
(49)                sqle.printStackTrace();
(50)            }
(51)        }
(52)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Wird der benötigte ResultSet geeignet (d.h. mit den Parameter CONCUR_UPDATABLE) (siehe Zeile 49) initialisiert so können Änderungen, die im Hauptspeicher durch die JDBC-API durchgeführt werden in die Datenbank persistiert werden.

Beispiel 15 zeigt dies exemplarisch für den Einfügevorgang eines neuen Tupels.

Die Voraussetzungen für Einfüge- und Aktualisierungsvorgänge entsprechen denen von *updatable* views, d.h. die Daten dürfen nur aus genau einer Tabelle entnommen sein und müssen den Primärschlüssel enthalten.

Beispiel 15: Auslesen und Einfügen von Daten

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.ResultSetMetaData;
(4)import java.sql.SQLException;
(5)import java.sql.Statement;
(6)
(7)import com.mysql.jdbc.Connection;
(8)
(9)public class JDBCSelect2 {
(10)    private static void printResultSet(ResultSet rs) throws SQLException {
(11)        ResultSetMetaData rsmd = rs.getMetaData();
(12)        int noColumns = rsmd.getColumnCount();
(13)        for (int i = 1; i < noColumns; i++) {
(14)            System.out.print(rsmd.getColumnLabel(i) + "\t");
(15)        }
(16)        System.out.println();
(17)
(18)        while (rs.isLast() == false) {
(19)            rs.next();
(20)            for (int i = 1; i < noColumns; i++) {
(21)                System.out.print( rs.getObject(i)+"\t" );
(22)            }
(23)            System.out.println();
(24)        }
(25)    }
(26)
(27)    public static void main(String[] args) {
(28)        try {
(29)            Class.forName("com.mysql.jdbc.Driver");
(30)        } catch (ClassNotFoundException e) {
(31)            System.err.println("Driver class not found");
(32)            e.printStackTrace();
(33)        }
(34)        Connection con = null;
(35)
(36)        try {
(37)            con =
(38)                (Connection) DriverManager.getConnection(
(39)                    "jdbc:mysql://localhost/jdbctest/",
(40)                    "mario",
(41)                    "thePassword");
(42)        } catch (SQLException e1) {
(43)            System.err.println("Error establishing database connection");
(44)            e1.printStackTrace();
(45)        }

```



```

(46)
(47)         Statement stmt = null;
(48)         try {
(49)             stmt = (Statement) con.createStatement(ResultSet.
TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
(50)         } catch (SQLException e2) {
(51)             System.err.println("Error creating SQL-Statement");
(52)             e2.printStackTrace();
(53)         }
(54)
(55)         try {
(56)             ResultSet uprs = (ResultSet) stmt.executeQuery("SELECT *
FROM EMPLOYEE;");
(57)             printResultSet(uprs);
(58)             uprs.moveToInsertRow();
(59)             uprs.updateString("FNAME", "Mario");
(60)             uprs.updateString("LNAME", "Jeckle");
(61)             uprs.updateInt("SSN", 11111111);
(62)             uprs.insertRow();
(63)             uprs = (ResultSet) stmt.executeQuery("SELECT * FROM
EMPLOYEE;");
(64)             printResultSet(uprs);
(65)         } catch (SQLException e3) {
(66)             System.err.println("Error selecting values from table
EMPLOYEE");
(67)             e3.printStackTrace();
(68)         }
(69)     }
(70)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Auf dieselbe Weise können auch Tupel einer Relation verändert werden. Hierzu stehen eine Reihe von updateXXX-Methoden zur Verfügung, wobei XXX für den Typ des zu aktualisierenden Attributs steht.

Nach durchgeführter Modifikation der hauptspeicherresidenten Werte werden diese durch [updateRow](#) in die Datenbank rückgeschrieben.

Beispiel 16 zeigt dies:

Beispiel 16: Modifizieren von Daten

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import java.sql.Statement;
(5)
(6)import com.mysql.jdbc.Connection;
(7)
(8)public class JDBCSelect3 {
(9)    public static void main(String[] args) {
(10)        try {
(11)            Class.forName("com.mysql.jdbc.Driver");
(12)        } catch (ClassNotFoundException e) {
(13)            System.err.println("Driver class not found");
(14)            e.printStackTrace();
(15)        }
(16)        Connection con = null;
(17)
(18)        try {
(19)            con =
(20)                (Connection) DriverManager.getConnection(
(21)                    "jdbc:mysql://localhost/jdbctest/",
(22)                    "mario",
(23)                    "thePassword");
(24)        } catch (SQLException e1) {
(25)            System.err.println("Error establishing database connection");
(26)            e1.printStackTrace();
(27)        }
(28)
(29)        Statement stmt = null;

```



```

(30)         try {
(31)             stmt =
(32)                 (Statement) con.createStatement(
(33)                     ResultSet.TYPE_SCROLL_SENSITIVE,
(34)                     ResultSet.CONCUR_UPDATABLE);
(35)         } catch (SQLException e2) {
(36)             System.err.println("Error creating SQL-Statement");
(37)             e2.printStackTrace();
(38)         }
(39)
(40)         try {
(41)             ResultSet uprs =
(42)                 (ResultSet) stmt.executeQuery("SELECT * FROM
EMPLOYEE;");
(43)             int namePos = uprs.findColumn("LNAME");
(44)
(45)             while (uprs.isLast() == false) {
(46)                 uprs.next();
(47)                 if (uprs.getString(namePos).compareTo("Wallace") ==
0) {
(48)                     uprs.updateString(namePos, "Doe");
(49)                     uprs.updateRow();
(50)                 }
(51)             }
(52)
(53)         } catch (SQLException e3) {
(54)             System.err.println("Error selecting values from table
EMPLOYEE");
(55)             e3.printStackTrace();
(56)         }
(57)     }
(58)}

```

[Download des Beispiels](#)

Analog vollzieht sich der Löschvorgang mittels [deleteRow](#):

Beispiel 17: Löschen von Daten

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import java.sql.Statement;
(5)
(6)import com.mysql.jdbc.Connection;
(7)
(8)public class JDBCSelect4 {
(9)    public static void main(String[] args) {
(10)        try {
(11)            Class.forName("com.mysql.jdbc.Driver");
(12)        } catch (ClassNotFoundException e) {
(13)            System.err.println("Driver class not found");
(14)            e.printStackTrace();
(15)        }
(16)        Connection con = null;
(17)
(18)        try {
(19)            con =
(20)                (Connection) DriverManager.getConnection(
(21)                    "jdbc:mysql://localhost/jdbctest/",
(22)                    "mario",
(23)                    "thePassword");
(24)        } catch (SQLException e1) {
(25)            System.err.println("Error establishing database connection");
(26)            e1.printStackTrace();
(27)        }
(28)
(29)        Statement stmt = null;
(30)        try {
(31)            stmt =
(32)                (Statement) con.createStatement(
(33)                    ResultSet.TYPE_SCROLL_SENSITIVE,

```



```

(34)                                ResultSet.CONCUR_UPDATABLE);
(35)    } catch (SQLException e2) {
(36)        System.err.println("Error creating SQL-Statement");
(37)        e2.printStackTrace();
(38)    }
(39)
(40)    try {
(41)        ResultSet uprs =
(42)            (ResultSet) stmt.executeQuery("SELECT * FROM
EMPLOYEE;");
(43)        int namePos = uprs.findColumn("LNAME");
(44)
(45)        while (uprs.isLast() == false) {
(46)            uprs.next();
(47)            if (uprs.getString(namePos).compareTo("Smith") == 0)
{
(48)                uprs.deleteRow();
(49)            }
(50)        }
(51)
(52)    } catch (SQLException e3) {
(53)        System.err.println("Error selecting values from table
EMPLOYEE");
(54)        e3.printStackTrace();
(55)    }
(56) }
(57)}

```

[Download des Beispiels](#)

Die bisher betrachteten Varianten extrahieren Daten aus der Datenbank im Stile einer Momentaufnahme (*snapshot*) zum Zeitpunkt der Anfrage. Die einmal angefragten Inhalte können sich jedoch noch zur Laufzeit der zugreifenden JDBC-Applikation datenbankseitig ändern, wenn sie durch eine andere Applikation neu geschrieben werden. Zur Gewährleistung der Konsistenz des extrahierten Snapshots mit den tatsächlichen Datenbankinhalten steht die Operation [rowUpdated](#) zur Verfügung. Sie ermittelt ob der im Hauptspeicher befindliche Wert mit dem aktuellen Datenbankinhalt übereinstimmt, d.h. ob der DB-Inhalt aktualisiert wurde.

Beispiel 18 zeigt ein Umsetzungsbeispiel.

Beispiel 18: Test auf geänderte Daten

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import com.mysql.jdbc.Connection;
(5)import com.mysql.jdbc.Statement;
(6)
(7)public class JDBCSelect7 {
(8)    public static void main(String[] args) {
(9)        try {
(10)            Class.forName("com.mysql.jdbc.Driver");
(11)        } catch (ClassNotFoundException cnfe) {
(12)            System.err.println("Driver class not found");
(13)            cnfe.printStackTrace();
(14)        }
(15)        Connection con = null;
(16)
(17)        try {
(18)            con =
(19)                (Connection) DriverManager.getConnection(
(20)                    "jdbc:mysql://localhost/jdbctest/",
(21)                    "mario",
(22)                    "thePassword");
(23)        } catch (SQLException sqle) {
(24)            System.err.println("Error establishing database connection");
(25)            sqle.printStackTrace();
(26)        }
(27)
(28)        Statement stmt = null;
(29)        try {
(30)            stmt =

```




```

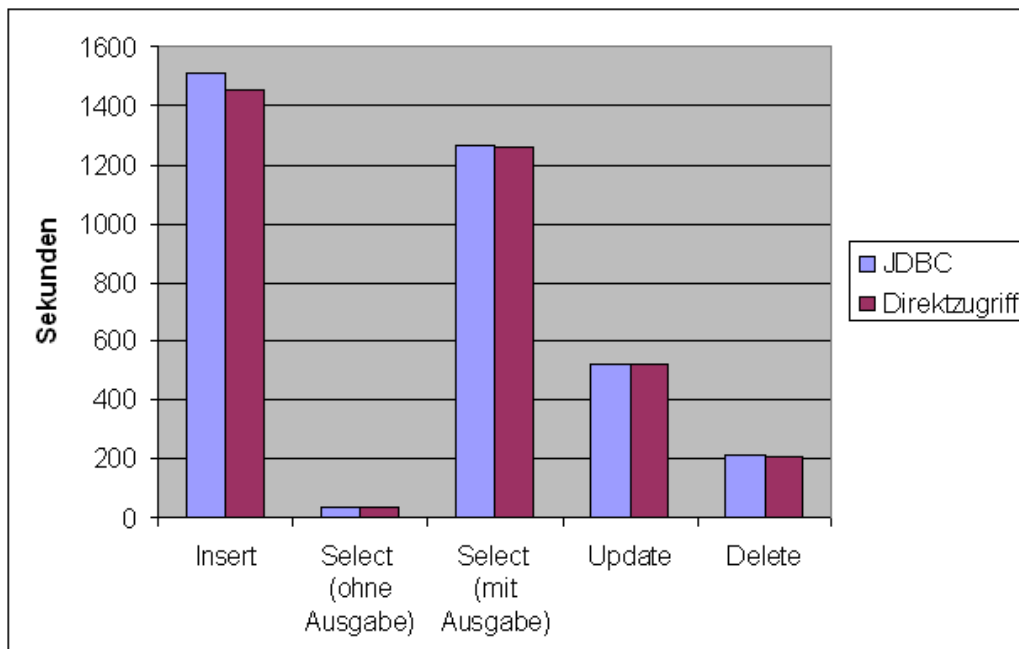
(31)                                (Statement) con.createStatement(
(32)                                ResultSet.TYPE_SCROLL_SENSITIVE,
(33)                                ResultSet.CONCUR_UPDATABLE);
(34)                                } catch (SQLException sqle) {
(35)                                    System.err.println("Error creating SQL-Statement");
(36)                                    sqle.printStackTrace();
(37)                                }
(38)
(39)                                try {
(40)                                    ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(41)                                    rs.absolute(5);
(42)                                    System.out.println(rs.getString("FNAME"));
(43)
(44)                                    System.out.println("sleeping ...");
(45)                                    Thread.sleep(6000);
(46)                                    System.out.println("awake ...");
(47)
(48)                                    if (rs.rowUpdated() == true) {
(49)                                        rs.refreshRow();
(50)                                        System.out.println(rs.getString("FNAME"));
(51)                                    }
(52)
(53)                                } catch (SQLException sqle) {
(54)                                    System.err.println("Error selecting values from table
EMPLOYEE");
(55)                                    sqle.printStackTrace();
(56)                                } catch (InterruptedException ie) {
(57)                                    ie.printStackTrace();
(58)                                }
(59)                                }
(60)}

```

[Download des Beispiels](#)

Performancebetrachtungen

Abbildung 3: JDBC-Geschwindigkeitsvergleich



(click on image to enlarge!)

Die Abbildung zeigt die Ergebnisse einiger Geschwindigkeitsmessungen als Vergleich zwischen dem Zugriff auf eine MySQL-Datenbank unter Nutzung der Textschnittstelle und der Abwicklung derselben Zugriffe mittels JDBC.

Zur Messung wurde eine nicht-indexierte Datenbank mit 10^7 Einträgen verwendet die aus der Relation tab bestand. Deren Tupel wurden aus Paaren von 36-Byte großen UUIDs gemäß dem [Spezifikationsentwurf der IETF](#) gebildet.

Zur Zeitmessung wurden folgende Einzeloperationen betrachtet:

- **Insert:** INSERT INTO tab VALUES(...)
Die Werte wurden unter Deaktivierung des Autocommit sequentiell eingefügt.
- **Select (ohne Ausgabe):** SELECT COUNT(*) FROM tab
Die ermittelte Gesamtzahl wurde nicht am Bildschirm ausgegeben.
- **Select (mit Ausgabe):** SELECT * FROM tab
Die Resultate wurden durch den Standardclient bzw. eine selbsterstellte (textmode) Java-Implementierung ausgegeben.
- **Update:** UPDATE tab SET UUID1="X" WHERE UUID1<>"X"
Durch die Initialisierung der Werte mit UUID-Einträgen wird sichergestellt, daß alle Tupel aktualisiert werden, da sie in keinem Fall den Wert X enthalten.
- **Delete:** DELETE FROM tab WHERE UUID2<>"X"
Durch die Initialisierung der Werte mit UUID-Einträgen wird sichergestellt, daß alle Tupel aktualisiert werden, da sie in keinem Fall den Wert X enthalten.

Insgesamt zeigt sich ein ausgewogenes Bild, in welchem der JDBC-Zugriff lediglich bei datenintensiven Zugriffen (große Mengen schreibender Zugriffe bei INSERT bzw. große Mengen lesender Operationen bei SELECT) im Bereich von fünf Prozent zurückliegt.

Diese enge Vergleichbarkeit der beiden Zugriffsmodi rührt von den Realisierung des eingesetzten JDBC-Treibers her; insbesondere von der Handhabung der physischen Datenbankverbindung auf Ebene des Netzwerkprotokolls.

SQL3-Datentypen

Die JDBC-API unterstützt mit Zugriffsmethoden auf die Datentypen BLOB, CLOB, ARRAY, Object und Ref bereits eine Untermenge des [SQL:1999-Standards](#). So können, vorausgesetzt das durch JDBC angesprochene DBMS unterstützt dies große unstrukturierte Binär- oder Textdaten sowie einfache verschachtelte Tabellen, mithin [NF²-Strukturen](#) verwaltet werden.

Beispiel 19 zeigt den Zugriff auf ein als eingebettete Tabelle realisiertes mengenwertiges Attribut. Die Beispieldatenbank wurde hierfür wie folgt modifiziert:

```
alter table EMPLOYEE ADD CAR SET('53M91','521R4', 'LL0415', 'XNU457');
update EMPLOYEE set CAR='XNU457' where SSN=123456789;
update EMPLOYEE set CAR='XNU457,521R4' where SSN="999887777";
```

Beispiel 19: Zugriff auf ein mengenwertiges Attribut

```
(1)import java.sql.Array;
(2)import java.sql.DriverManager;
(3)import java.sql.ResultSet;
(4)import java.sql.SQLException;
(5)import com.mysql.jdbc.Connection;
(6)import com.mysql.jdbc.Statement;
(7)
(8)public class JDBCSelect8 {
(9)    public static void main(String[] args) {
(10)        try {
(11)            Class.forName("com.mysql.jdbc.Driver");
(12)        } catch (ClassNotFoundException cnfe) {
(13)            System.err.println("Driver class not found");
(14)            cnfe.printStackTrace();
(15)        }
(16)        Connection con = null;
(17)
(18)        try {
(19)            con =
(20)                (Connection) DriverManager.getConnection(
(21)                    "jdbc:mysql://localhost/jdbctest/",
(22)                    "mario",
(23)                    "thePassword");
(24)        } catch (SQLException sqle) {
(25)            System.err.println("Error establishing database connection");
(26)            sqle.printStackTrace();
(27)        }
(28)
(29)        Statement stmt = null;
(30)        try {
(31)            stmt = (Statement) con.createStatement();
```



```

(32)    } catch (SQLException sqle) {
(33)        System.err.println("Error creating SQL-Statement");
(34)        sqle.printStackTrace();
(35)    }
(36)
(37)    try {
(38)        ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(39)        while (!rs.isLast()) {
(40)            rs.first();
(41)            System.out.print(rs.getString("FNAME") + "\t");
(42)            Array cars = rs.getArray("CAR");
(43)            ResultSet carsRS = cars.getResultSet();
(44)            System.out.print("(");
(45)            while (!carsRS.isLast()) {
(46)                rs.first();
(47)                System.out.print(carsRS.getString("CAR"));
(48)                carsRS.next();
(49)            }
(50)            System.out.println(")");
(51)            rs.next();
(52)        }
(53)    } catch (SQLException sqle) {
(54)        System.err.println("Error selecting values from table
EMPLOYEE");
(55)        sqle.printStackTrace();
(56)    }
(57) }
(58)}

```

[Download des Beispiels](#)

Das Beispiel unterstreicht die Rolle der mengenwertigen Attribute als eingebettete Tabellen. So erfolgt der Zugriff auf die Einzelwerte des Attributs CAR identisch zur Ermittlung der Resultatmenge der SQL-Anfrage mittels `getResultSet`. Auch die Traversierung der einzelnen CAR-Elemente erfolgt äquivalent.

Die Aufnahme der *large objects* in ihrer Ausprägungsform als *Character Large Objects* (CLOB) oder *Binary Large Objects* (BLOB) stellen eine der zentralen Erweiterungen des SQL:1999-Standards gegenüber seinen Vorgängern dar.

Zwar ist die Ablage großer unstrukturierter Datenobjekte in relationalen Datenbanken konzeptionell durchaus diskussionswert, jedoch in der Praxis oftmals, trotz der teilweise erheblichen Geschwindigkeitseinbußen im Zugriff (so benötigt die Ausführung der Beispielapplikation mit einem 10^6 Byte großen Datenstrom 1,1 Sekunden, während dieselbe Operation dateisystembasiert in 0,1 Sekunde abläuft), gewünscht.

Beispiel 20 zeigt die notwendigen Schritte zur Ablage und erneuten Auslese eines aus einer Datei gewonnen Binärdatenstroms in der Datenbank.

Die Beispieldatenbank wurde hierfür um ein Attribut zur Aufnahme binärer Daten erweitert:

```
ALTER TABLE EMPLOYEE ADD binData blob;
```

Beispiel 20: Verarbeitung unstrukturierter Binärdaten

```

(1)import java.io.File;
(2)import java.io.FileInputStream;
(3)import java.io.FileOutputStream;
(4)import java.io.IOException;
(5)import java.sql.DriverManager;
(6)import java.sql.PreparedStatement;
(7)import java.sql.ResultSet;
(8)import java.sql.SQLException;
(9)
(10)import com.mysql.jdbc.Connection;
(11)import com.mysql.jdbc.Statement;
(12)
(13)public class JDBCSelect9 {
(14)    public static void main(String[] args) {
(15)        try {
(16)            Class.forName("com.mysql.jdbc.Driver");
(17)        } catch (ClassNotFoundException cnfe) {
(18)            System.err.println("Driver class not found");
(19)            cnfe.printStackTrace();
(20)        }

```



```

(21)         Connection con = null;
(22)
(23)         try {
(24)             con =
(25)                 (Connection) DriverManager.getConnection(
(26)                     "jdbc:mysql://localhost/jdbctest/",
(27)                     "mario",
(28)                     "thePassword");
(29)         } catch (SQLException sqle) {
(30)             System.err.println("Error establishing database connection");
(31)             sqle.printStackTrace();
(32)         }
(33)
(34)         try {
(35)             File file = new File(args[0]);
(36)             FileInputStream fis = new FileInputStream(args[0]);
(37)             PreparedStatement pstmt =
(38)                 con.prepareStatement(
(39)                     "UPDATE EMPLOYEE SET binData =? WHERE
SSN=123456789");
(40)             pstmt.setBinaryStream(1, fis, (int) file.length());
(41)             pstmt.executeUpdate();
(42)             fis.close();
(43)
(44)             //read it back from the database
(45)             Statement stmt = (Statement) con.createStatement();
(46)             ResultSet rs =
(47)                 stmt.executeQuery(
(48)                     "SELECT binData FROM EMPLOYEE WHERE
SSN='123456789'");
(49)
(50)             FileOutputStream fos = new FileOutputStream(args[1]);
(51)             if (rs.next())
(52)                 fos.write(rs.getBytes(1));
(53)             fos.close();
(54)
(55)         } catch (SQLException sqle) {
(56)             System.err.println("Error selecting values from table
EMPLOYEE");
(57)             sqle.printStackTrace();
(58)         } catch (IOException ioe) {
(59)             ioe.printStackTrace();
(60)         }
(61)     }
(62)}

```

[Download des Beispiels](#)

Die Binärdaten können naturgemäß nicht direkt in die SQL-UPDATE-Anweisung eingebunden werden, sie werden daher einer mittels `prepareStatement` vorerzeugten Anweisung durch Aufruf der Methode `setBinaryStream` übergeben.

Transaktionssteuerung

Zur Steuerung des transaktionalen Verhaltens einer JDBC-Anfrage bietet die Klasse `Connection` verschiedene Methoden an:

- Abfrage der aktuellen Isolationsstufe: `getIsolationLevel`.
Hierbei werden fünf Stufen unterschieden:
 - `TRANSACTION_NONE`: Keinerlei Transaktionsunterstützung
 - `TRANSACTION_READ_UNCOMMITTED`: Auch nicht durch commit freigegebene Daten werden gelesen.
Es können daher *dirty reads*, Nicht-wiederholbare- und Phantomlesevorgänge auftreten.
 - `TRANSACTION_READ_COMMITTED`: Nur durch commit freigegebene Daten werden gelesen.
nichtwiederholbare- und Phantomlesevorgänge können jedoch auftreten.
 - `TRANSACTION_REPEATABLE_READ`: Innerhalb einer Transaktion können die verarbeiteten Daten nicht durch eine andere Transaktion verändert werden. Das Auftreten von *dirty reads* und nichtwiederholbaren Lesevorgängen ist daher ausgeschlossen, Phantomlesevorgänge sind jedoch weiterhin möglich.
 - `TRANSACTION_SERIALIZABLE`: Strikte Isolation aller Transaktionen, auf dieser Stufe sind auch

Phantomlesevorgänge ausgeschlossen.

Jede Stufe geht zwar mit einer gesteigerten Qualität der durch eine Transaktion verarbeiteten Daten einher, jedoch senkt gleichzeitig eine striktere Isolationsstufe die Anzahl der gleichzeitigen Zugriffe auf die Datenbank und damit die Gesamtsystemperformance.

- Setzen der Isolationsstufe: [setTransactionIsolation](#)
- (De-)Aktivierung der automatischen Freigabe: [setAutoCommit](#). Die Übergabe von true bewirkt die Aktivierung des Modus bei dem jede Einzelanweisung sofort persistent übernommen und für andere Transaktionen sichtbar wird.
Standardmäßig ist diese Option aktiviert. Ihr aktueller Zustand kann per [getAutoCommit](#) ermittelt werden.
- Freigabe von Änderungen: [commit](#).
- Rücknahme von Änderungen: [rollback](#).
- Rücknahme von Änderungen bis zu definiertem Sicherungspunkt: [rollback\(Savepoint s\)](#).
- Setzen eines Sicherungspunktes: [setSavepoint](#).

Beispiel 1 zeigt die Nutzung des Transaktionskonzepts.

Zunächst wird die aktuelle Isolationsstufe ermittelt und geprüft ob das angesprochene DBMS die höchste durch JDBC vorhergesehene Isolationsstufe unterstützt.

Nach Abschaltung der automatischen Änderungsübernahme (`setAutoCommit(false)`) werden zunächst zwei Tupel in die Tabelle EMPLOYEE eingefügt, die jedoch nur innerhalb der laufenden Transaktion sichtbar werden, für alle anderen Transaktionen innerhalb des DBMS bleiben die neuen Werte (zunächst) unsichtbar.

Eine angenommene Fehlersituation führt zum Rücksetzen der Transaktion durch (`rollback`).

Nach Abschluß des Programms wurden zwar die beiden ersten Werte lokal in die Datenbank übernommen, aber noch innerhalb der laufenden Transaktion wieder daraus entfernt, weshalb sie zu keinem Zeitpunkt für andere Datenbankbenutzer sichtbar waren.

Beispiel 21: Transaktionsverarbeitung

```
(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)
(5)import com.mysql.jdbc.Connection;
(6)import com.mysql.jdbc.Statement;
(7)
(8)public class JDBCTransact1 {
(9)    private static void printContent(Statement stmt) throws SQLException {
(10)        ResultSet rs =
(11)            stmt.executeQuery("SELECT FNAME,MINIT,LNAME FROM EMPLOYEE;");
(12)        while (!rs.isLast()) {
(13)            rs.next();
(14)            System.out.println(
(15)                rs.getString("LNAME")
(16)                + "\t"
(17)                + rs.getString("MINIT")
(18)                + "\t"
(19)                + rs.getString("LNAME"));
(20)        }
(21)    }
(22)    public static void main(String[] args) {
(23)        try {
(24)            Class.forName("com.mysql.jdbc.Driver");
(25)        } catch (ClassNotFoundException cnfe) {
(26)            System.err.println("Driver class not found");
(27)            cnfe.printStackTrace();
(28)        }
(29)        Connection con = null;
(30)
(31)        try {
(32)            con =
(33)                (Connection) DriverManager.getConnection(
(34)                    "jdbc:mysql://localhost/jdbctest/",
(35)                    "mario",
(36)                    "thePassword");
(37)        } catch (SQLException sqle) {
(38)            System.err.println("Error establishing database connection");
(39)            sqle.printStackTrace();
(40)        }
(41)
(42)        Statement stmt = null;
(43)        try {
```



```

(44)         stmt = (Statement) con.createStatement();
(45)     } catch (SQLException sqle) {
(46)         System.err.println("Error creating SQL-Statement");
(47)         sqle.printStackTrace();
(48)     }
(49)
(50)     try {
(51)         int transactionIsolation = con.getTransactionIsolation();
(52)         switch (transactionIsolation) {
(53)             case Connection.TRANSACTION_NONE :
(54)                 System.out.println("Transactions are not
supported");
(55)                 break;
(56)             case Connection.TRANSACTION_READ_UNCOMMITTED :
(57)                 System.out.println(
(58)                     "Dirty reads, non-repeatable reads
and phantom reads can occur");
(59)                 break;
(60)             case Connection.TRANSACTION_READ_COMMITTED :
(61)                 System.out.println(
(62)                     "Dirty reads are prevented; non-
repeatable reads and phantom reads can occur");
(63)                 break;
(64)             case Connection.TRANSACTION_REPEATABLE_READ :
(65)                 System.out.println(
(66)                     "Dirty reads and non-repeatable
reads are prevented; phantom reads can occur");
(67)                 break;
(68)             case Connection.TRANSACTION_SERIALIZABLE :
(69)                 System.out.println(
(70)                     "Dirty reads, non-repeatable reads
and phantom reads are prevented");
(71)                 break;
(72)             }
(73)             if (transactionIsolation < Connection.
TRANSACTION_SERIALIZABLE) {
(74)                 con.setTransactionIsolation(
(75)                     Connection.TRANSACTION_SERIALIZABLE);
(76)                 if (con.getTransactionIsolation()
(77)                     != Connection.TRANSACTION_SERIALIZABLE) {
(78)                     System.out.println(
(79)                         "cannot set Connection.
TRANSACTION_SERIALIZABLE");
(80)                 } else {
(81)                     System.out.println(
(82)                         "reached highest possible isolation
level");
(83)                 }
(84)             }
(85)
(86)             con.setAutoCommit(false);
(87)             stmt.executeUpdate(
(88)                 "INSERT INTO EMPLOYEE VALUES
('Hans', 'X', 'Hinterhuber', '11111111', NULL, NULL, NULL, NULL, NULL, NULL);");
(89)             stmt.executeUpdate(
(90)                 "INSERT INTO EMPLOYEE VALUES
('Franz', 'X', 'Obermüller', '22222222', NULL, NULL, NULL, NULL, NULL, NULL);");
(91)             printContent(stmt);
(92)             //suppose error happens here
(93)             Thread.sleep(5000);
(94)             boolean error = true;
(95)             if (error) {
(96)                 con.rollback();
(97)             } else {
(98)                 stmt.executeUpdate(
(99)                     "INSERT INTO EMPLOYEE VALUES
('Fritz', 'X', 'Meier', '33333333', NULL, NULL, NULL, NULL, NULL, NULL);");
(100)            }
(101)            printContent(stmt);
(102)        } catch (SQLException sqle) {
(103)            sqle.printStackTrace();
(104)        } catch (InterruptedException ie) {

```

```

(105)                ie.printStackTrace();
(106)            }
(107)    }
(108)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Neben dem Zurücksetzen einer vollständigen Transaktion bietet die JDBC-API auch die Möglichkeit alle Schritte bis zu einem anwenderdefinierten aus der Datenbank zu entfernen.

Beispiel 22 zeigt dies unter Verwendung der Methode `setSavepoint` zur Definition eines Sicherungspunktes und `rollback(sp)` zum Zurücksetzen bis zu diesem Sicherungspunkt.

Beispiel 22: Transaktionsverarbeitung mit Sicherungspunkten

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import java.sql.Savepoint;
(5)
(6)import com.mysql.jdbc.Connection;
(7)import com.mysql.jdbc.Statement;
(8)
(9)public class JDBCTransact2 {
(10)    private static void printContent(Statement stmt) throws SQLException {
(11)        ResultSet rs =
(12)            stmt.executeQuery("SELECT FNAME,MINIT,LNAME FROM EMPLOYEE;");
(13)        while (!rs.isLast()) {
(14)            rs.next();
(15)            System.out.println(
(16)                rs.getString("LNAME")
(17)                    + "\t"
(18)                    + rs.getString("MINIT")
(19)                    + "\t"
(20)                    + rs.getString("LNAME"));
(21)        }
(22)    }
(23)    public static void main(String[] args) {
(24)        try {
(25)            Class.forName("com.mysql.jdbc.Driver");
(26)        } catch (ClassNotFoundException cnfe) {
(27)            System.err.println("Driver class not found");
(28)            cnfe.printStackTrace();
(29)        }
(30)        Connection con = null;
(31)
(32)        try {
(33)            con =
(34)                (Connection) DriverManager.getConnection(
(35)                    "jdbc:mysql://localhost/jdbctest/",
(36)                    "mario",
(37)                    "thePassword");
(38)        } catch (SQLException sqle) {
(39)            System.err.println("Error establishing database connection");
(40)            sqle.printStackTrace();
(41)        }
(42)
(43)        Statement stmt = null;
(44)        try {
(45)            stmt = (Statement) con.createStatement();
(46)        } catch (SQLException sqle) {
(47)            System.err.println("Error creating SQL-Statement");
(48)            sqle.printStackTrace();
(49)        }
(50)
(51)        try {
(52)            int transactionIsolation = con.getTransactionIsolation();
(53)            switch (transactionIsolation) {
(54)                case Connection.TRANSACTION_NONE :
(55)                    System.out.println("Transactions are not
supported");
(56)                    break;

```




```

(57)                                case Connection.TRANSACTION_READ_UNCOMMITTED :
(58)                                System.out.println(
(59)                                "Dirty reads, non-repeatable reads
and phantom reads can occur");
(60)                                break;
(61)                                case Connection.TRANSACTION_READ_COMMITTED :
(62)                                System.out.println(
(63)                                "Dirty reads are prevented; non-
repeatable reads and phantom reads can occur");
(64)                                break;
(65)                                case Connection.TRANSACTION_REPEATABLE_READ :
(66)                                System.out.println(
(67)                                "Dirty reads and non-repeatable
reads are prevented; phantom reads can occur");
(68)                                break;
(69)                                case Connection.TRANSACTION_SERIALIZABLE :
(70)                                System.out.println(
(71)                                "Dirty reads, non-repeatable reads
and phantom reads are prevented");
(72)                                break;
(73)                                }
(74)                                if (transactionIsolation < Connection.
TRANSACTION_SERIALIZABLE) {
(75)                                con.setTransactionIsolation(
(76)                                Connection.TRANSACTION_SERIALIZABLE);
(77)                                if (con.getTransactionIsolation()
(78)                                != Connection.TRANSACTION_SERIALIZABLE) {
(79)                                System.out.println(
(80)                                "cannot set Connection.
TRANSACTION_SERIALIZABLE");
(81)                                } else {
(82)                                System.out.println(
(83)                                "reached highest possible isolation
level");
(84)                                }
(85)                                }
(86)
(87)                                con.setAutoCommit(false);
(88)                                stmt.executeUpdate(
(89)                                "INSERT INTO EMPLOYEE VALUES
('Hans', 'X', 'Hinterhuber', '11111111', NULL, NULL, NULL, NULL, NULL);");
(90)                                Savepoint sp = con.setSavepoint();
(91)                                stmt.executeUpdate(
(92)                                "INSERT INTO EMPLOYEE VALUES
('Franz', 'X', 'Obermüller', '22222222', NULL, NULL, NULL, NULL, NULL);");
(93)                                printContent(stmt);
(94)                                //suppose error happens here
(95)                                Thread.sleep(5000);
(96)                                boolean error = true;
(97)                                if (error) {
(98)                                con.rollback(sp);
(99)                                }
(100)                               stmt.executeUpdate(
(101)                               "INSERT INTO EMPLOYEE VALUES
('Fritz', 'X', 'Meier', '33333333', NULL, NULL, NULL, NULL, NULL);");
(102)                               printContent(stmt);
(103)                               con.commit();
(104)                               } catch (SQLException sqle) {
(105)                               sqle.printStackTrace();
(106)                               } catch (InterruptedException ie) {
(107)                               ie.printStackTrace();
(108)                               }
(109)       }
(110) }

```

[Download des Beispiels](#)

Netzwerkverkehr

Der Netzwerkmitschnitt zeigt den TCP-Kommunikationsverlauf der SQL-Anfrage `SELECT * FROM`

<http://www.jeckle.de/vorlesung/DB-Anwendungen/script.html> (31 of 76)09.01.2004 07:38:59

```

0420 b 03 Y E S 0e 00 00 1d \t h a v e _ i s a m 03 Y E S \r 00 00 1e
\t h a v e
0440 _ r a i d 02 N O 16 00 00 1f \f h a v e _ s y m l i n k \b D
I S A B L
0460 E D 10 00 00 \f h a v e _ o p e n s s l 02 N O 15 00 00 ! 10
h a v e _
0480 q u e r y _ c a c h e 03 Y E S 0b 00 00 " \t i n i t _ f i
l l e 00 ( 00
04a0 00 # 1f i n n o d b _ a d d i t i o n a l _ m e m _ p o
o l _ s i
04c0 z e 07 2 0 9 7 1 5 2 ! 00 00 $ 17 i n n o d b _ b u f f e
r _ p o o
04e0 l _ s i z e \b 1 6 7 7 7 2 1 6 - 00 00 % 15 i n n o d b _
d a t a _
0500 f i l e _ p a t h 16 i b d a t a 1 : 1 0 M : a u t o e
x t e n d
0520 16 00 00 & 14 i n n o d b _ d a t a _ h o m e _ d i r 00 19
00 00 ' 16 i
0540 n n o d b _ f i l e _ i o _ t h r e a d s 01 4 18 00 00
( 15 i n n o
0560 d b _ f o r c e _ r e c o v e r y 01 0 1c 00 00 ) 19 i n n
o d b _ t
0580 h r e a d _ c o n c u r r e n c y 01 8 ! 00 00 * 1e i n n
o d b _ f
05a0 l u s h _ l o g _ a t _ t r x _ c o m m i t 01 1 18 00 00
+ 14 i n n
05c0 o d b _ f a s t _ s h u t d o w n 02 0 N 15 00 00 , 13 i n
n o d b _
05e0 f l u s h _ m e t h o d 00 1c 00 00 - 18 i n n o d b _ l o
c k _ w a
0600 i t _ t i m e o u t 02 5 0 17 00 00 . 13 i n n o d b _ l o
g _ a r c
0620 h _ d i r 02 . / 17 00 00 / 12 i n n o d b _ l o g _ a r c
h i v e 03
0640 0 F F 1f 00 00 0 16 i n n o d b _ l o g _ b u f f e r _ s
i z e 07 8
0660 3 8 8 6 0 8 1d 00 00 1 14 i n n o d b _ l o g _ f i l e _
s i z e 07
0680 5 2 4 2 8 8 0 1c 00 00 2 19 i n n o d b _ l o g _ f i l e
s _ i n _
06a0 g r o u p 01 2 1d 00 00 3 19 i n n o d b _ l o g _ g r o u
p _ h o m
06c0 e _ d i r 02 . / 1d 00 00 4 1a i n n o d b _ m i r r o r e
d _ l o g
06e0 _ g r o u p s 01 1 1a 00 00 5 13 i n t e r a c t i v e _ t
i m e o u
0700 t 05 2 8 8 0 0 18 00 00 6 10 j o i n _ b u f f e r _ s i z
e 06 1 3 1
0720 0 7 2 19 00 00 7 0f k e y _ b u f f e r _ s i z e \b 1 6 7
7 7 2 1 6
0740 L 00 00 8 \b l a n g u a g e B / o p t / r a i d / m y s
q l - s t
0760 a n d a r d - 4 . 0 . 1 2 - p c - l i n u x - i 6 8
6 / s h a r
0780 e / m y s q l / e n g l i s h / 17 00 00 9 13 l a r g e _
f i l e s
07a0 _ s u p p o r t 02 0 N 10 00 00 : \f l o c a l _ i n f i l
e 02 0 N 15
07c0 00 00 ; 10 l o c k e d _ i n _ m e m o r y 03 0 F F \b 00 00
< 03 l o g
07e0 03 0 F F 0f 00 00 = \n l o g _ u p d a t e 03 0 F F 0b 00 00 >
07 l o g _
0800 b i n 02 0 N 16 00 00 ? 11 l o g _ s l a v e _ u p d a t e
s 03 0 F F
0820 15 00 00 @ 10 l o g _ s l o w _ q u e r i e s 03 0 F F 11 00
00 A \f l o
0840 g _ w a r n i n g s 03 0 F F 13 00 00 B 0f l o n g _ q u e
r y _ t i
0860 m e 02 1 0 19 00 00 C 14 l o w _ p r i o r i t y _ u p d a
t e s 03 0
0880 F F 1b 00 00 D 16 l o w e r _ c a s e _ t a b l e _ n a m
e s 03 0 F

```

```

08a0 F 1b 00 00 E 12 m a x _ a l l o w e d _ p a c k e t 07 1 0
4 7 5 5 2
08c0 ! 00 00 F 15 m a x _ b i n l o g _ c a c h e _ s i z e \n
4 2 9 4 9
08e0 6 7 2 9 5 1b 00 00 G 0f m a x _ b i n l o g _ s i z e \n 1
0 7 3 7 4
0900 1 8 2 4 14 00 00 H 0f m a x _ c o n n e c t i o n s 03 1 0
0 16 00 00 I
0920 12 m a x _ c o n n e c t _ e r r o r s 02 1 0 17 00 00 J 13
m a x _ d
0940 e l a y e d _ t h r e a d s 02 2 0 1d 00 00 K 13 m a x _ h
e a p _ t
0960 a b l e _ s i z e \b 1 6 7 7 7 2 1 6 19 00 00 L \r m a x _
j o i n _
0980 s i z e \n 4 2 9 4 9 6 7 2 9 5 15 00 00 M 0f m a x _ s o r
t _ l e n
09a0 g t h 04 1 0 2 4 17 00 00 N 14 m a x _ u s e r _ c o n n e
c t i o n
09c0 s 01 0 12 00 00 0 0e m a x _ t m p _ t a b l e s 02 3 2 00
00 P 14 m a
09e0 x _ w r i t e _ l o c k _ c o u n t \n 4 2 9 4 9 6 7 2
9 5 * 00 00
0a00 Q 1f m y i s a m _ m a x _ e x t r a _ s o r t _ f i l
e _ s i z
0a20 e \t 2 6 8 4 3 5 4 5 6 % 00 00 R 19 m y i s a m _ m a x _
s o r t _
0a40 f i l e _ s i z e \n 2 1 4 7 4 8 3 6 4 7 1b 00 00 S 16 m y
i s a m _
0a60 r e c o v e r _ o p t i o n s 03 0 F F 00 00 T 17 m y i
s a m _ s
0a80 o r t _ b u f f e r _ s i z e 07 8 3 8 8 6 0 8 17 00 00 U
11 n e t _
0aa0 b u f f e r _ l e n g t h 04 8 1 9 2 14 00 00 V 10 n e t _
r e a d _
0ac0 t i m e o u t 02 3 0 13 00 00 W 0f n e t _ r e t r y _ c o
u n t 02 1
0ae0 0 15 00 00 X 11 n e t _ w r i t e _ t i m e o u t 02 6 0 \b
00 00 Y 03 n
0b00 e w 03 0 F F 13 00 00 Z 10 o p e n _ f i l e s _ l i m i t
01 0 F 00 00
0b20 [ \b p i d _ f i l e < / o p t / r a i d / m y s q l -
s t a n d
0b40 a r d - 4 . 0 . 1 2 - p c - l i n u x - i 6 8 6 / d a
t a / l i
0b60 n u x . p i d 0b 00 00 \ \t l o g _ e r r o r 00 \n 00 00 ] 04
p o r t 04
0b80 3 3 0 6 14 00 00 ^ 10 p r o t o c o l _ v e r s i o n 02 1
0 18 00 00 _
0ba0 10 r e a d _ b u f f e r _ s i z e 06 1 3 1 0 7 2 1c 00 00
` 14 r e a
0bc0 d _ r n d _ b u f f e r _ s i z e 06 2 6 2 1 4 4 14 00 00
a 11 r p l
0be0 _ r e c o v e r y _ r a n k 01 0 1a 00 00 b 11 q u e r y _
c a c h e
0c00 _ l i m i t 07 1 0 4 8 5 7 6 13 00 00 c 10 q u e r y _ c a
c h e _ s
0c20 i z e 01 0 14 00 00 d 10 q u e r y _ c a c h e _ t y p e 02
0 N \f 00 00
0c40 e \t s e r v e r _ i d 01 1 17 00 00 f 11 s l a v e _ n e t
_ t i m e
0c60 o u t 04 3 6 0 0 19 00 00 g 15 s k i p _ e x t e r n a l _
l o c k i
0c80 n g 02 0 N 14 00 00 h 0f s k i p _ n e t w o r k i n g 03 0
F F 17 00 00
0ca0 i 12 s k i p _ s h o w _ d a t a b a s e 03 0 F F 13 00 00
j 10 s l o
0cc0 w _ l a u n c h _ t i m e 01 2 17 00 00 k 06 s o c k e t
0f / t m p /
0ce0 m y s q l . s o c k 18 00 00 l 10 s o r t _ b u f f e r _
s i z e 06
0d00 5 2 4 2 8 0 0b 00 00 m \b s q l _ m o d e 01 0 0f 00 00 n 0b t
a b l e _

```

```

0d20 c a c h e 02 6 4 12 00 00 o \n t a b l e _ t y p e 06 I N N
0 D B 14 00
0d40 00 p 11 t h r e a d _ c a c h e _ s i z e 01 0 14 00 00 q \f
t h r e a
0d60 d _ s t a c k 06 1 9 6 6 0 8 1d 00 00 r \f t x _ i s o l a
t i o n 0f
0d80 R E P E A T A B L E - R E A D 0e 00 00 s \b t i m e z o n
e 04 C E S
0da0 T 18 00 00 t 0e t m p _ t a b l e _ s i z e \b 3 3 5 5 4 4
3 2 \r 00 00
0dc0 u 06 t m p d i r 05 / t m p / 1c 00 00 v 07 v e r s i o n 13
4 . 0 . 1
0de0 2 - s t a n d a r d - l o g 13 00 00 w \f w a i t _ t i m
e o u t 05
0ee0 2 8 8 0 0 01 00 00 x p 11 00 00 00 03 S E T a u t o c o m m
i t = 1 03
0e20 00 00 01 00 00 00 18 00 00 00 03 S E L E C T * F R O M E M
P L O Y E
0e40 E ; 01 00 00 01 \n 19 00 00 02 \b E M P L O Y E E 05 F N A M E 03
\n 00 00 01 ý
0e60 03 01 00 00 19 00 00 03 \b E M P L O Y E E 05 M I N I T 03 01 00 00
01 p 03 00 00
0e80 00 19 00 00 04 \b E M P L O Y E E 05 L N A M E 03 \n 00 00 01 ý 03
01 00 00 17 00
0ea0 00 05 \b E M P L O Y E E 03 S S N 03 \t 00 00 01 03 03 01 00 00 19 00
00 06 \b E M
0ec0 P L O Y E E 05 B D A T E 03 \n 00 00 01 \n 03 00 00 00 1b 00 00 07 \b
E M P L O
0ee0 Y E E 07 A D D R E S S 03 1e 00 00 01 ý 03 00 00 00 17 00 00 \b \b E
M P L O Y
0f00 E E 03 S E X 03 01 00 00 01 p 03 00 01 00 1a 00 00 \t \b E M P L O Y
E E 06 S A
0f20 L A R Y 03 07 00 00 01 05 03 00 02 1c 00 00 \n \b E M P L O Y E E
\b S U P E
0f40 R S S N 03 \t 00 00 01 03 03 00 00 00 17 00 00 0b \b E M P L O Y E E
03 D N O 03
0f60 01 00 00 01 03 03 00 00 00 01 00 00 \f p R 00 00 \r 04 J o h n 01 B 05 S
m i t h \t
0f80 1 2 3 4 5 6 7 8 9 \n 1 9 6 5 - 0 1 - 0 9 18 7 3 1 F o
n d r e n
0fa0 , H o u s t o n , T X 01 M \b 3 0 0 0 0 . 0 0 \t 3 3
3 4 4 5 5
0fc0 5 5 01 5 R 00 00 0e \b F r a n k l i n 01 T 04 W o n g \t 3 3
3 4 4 5 5
0fe0 5 5 \n 1 9 5 5 - 1 2 - 0 8 15 6 3 8 V o s s , H o u
s t o n ,
1000 T X 01 M \b 4 0 0 0 0 . 0 0 \t 8 8 8 6 6 5 5 5 5 01 5 T
00 00 0f 06 A
1020 l i c i a 01 J 06 Z e l a y a \t 9 9 9 8 8 7 7 7 7 \n 1 9
6 8 - 0 7
1040 - 1 9 17 3 3 2 1 C a s t l e , S p r i n g , T X
01 F \b 2 5
1060 0 0 0 . 0 0 \t 9 8 7 6 5 4 3 2 1 01 4 W 00 00 10 \b J e n n
i f e r 01
1080 S 07 W a l l a c e \t 9 8 7 6 5 4 3 2 1 \n 1 9 4 1 - 0 6
- 2 0 17 2
10a0 9 1 B e r r y , B e l l a i r e , T X 01 F \b 4 3
0 0 0 . 0
10c0 0 \t 8 8 8 6 6 5 5 5 5 01 4 V 00 00 11 06 R a m e s h 01 K 07
N a r a y
10e0 a n \t 6 6 6 8 8 4 4 4 4 \n 1 9 6 2 - 0 9 - 1 5 18 9 7
5 F i r e
1100 0 a k , H u m b l e , T X 01 M \b 3 8 0 0 0 . 0 0
\t 3 3 3 4
1120 4 5 5 5 5 01 5 S 00 00 12 05 J o y c e 01 A 07 E n g l i s h
\t 4 5 3 4
1140 5 3 4 5 3 \n 1 9 7 2 - 0 7 - 3 1 16 5 6 3 1 R i c
e , H o u s
1160 t o n , T X 01 F \b 2 5 0 0 0 . 0 0 \t 3 3 3 4 4 5 5 5
5 01 5 S 00
1180 00 13 05 A h m a d 01 V 06 J a b b a r \t 9 8 7 9 8 7 9 8 7
\n 1 9 6 9

```

```

11a0 - 0 3 - 2 9 17 9 8 0   D a l l a s ,   H o u s t o
n ,   T X 01 M
11c0 \b 2 5 0 0 0 . 0 0 \t 9 8 7 6 5 4 3 2 1 01 4 6 00 00 14 05 J
a m e s 01
11e0 E 04 B o r g \t 8 8 8 6 6 5 5 5 5 \n 1 9 3 7 - 1 1 - 1 0
16 4 5 0
1200 S t o n e ,   H o u s t o n ,   T X 01 M \b 5 5 0 0 0 .
0 0 û 01 1
1220 01 00 00 15 þ

```

Die Analyse des Datenverkehrs zeigt, daß im Falle der JDBC-basierten Kommunikation ein gegenüber der nativen Schnittstelle um 3529 Byte vergrößertes Datenaufkommen ausgetauscht wird. Diese zusätzliche Datenmenge fällt jedoch nur einmal zum Zeitpunkt des JDBC-Verbindungsaufbaus statisch an. ([Vgl. Mitschnitt der mehrfachen Ausführung einer SQL-Anfrage innerhalb einer bestehenden JDBC-Verbindung](#))

Zusätzlich offenbart Zeile 0x40 des Datenverkehrs die verschlüsselte Übermittlung des Paßwortes des Anwenders *mario*. Allerdings werden die per Anfrage ermittelten Nutzdaten (ab Zeile 0xe40) unverschlüsselt über die Netzwerkschnittstelle übertragen und stellen somit ein potentiellies Angriffsziel dar.

Abhilfe hierfür kann die Tunnelung des Datenverkehrs, beispielsweise mittels [SSH](#), durch eine sichere Verbindung bieten.

Web-Referenzen 1: Weiterführende Links



- [JDBC @ SUN](#)
- [JDBC learning center @ SUN](#)
- [JDBC Tutorial](#)
- [JDBC FAQ @ JGuru.com](#)
- [G. Reese: Database Programming with JDBC and Java. O'Reilly, 1997](#)
- [Verhältnis von X/Open CLI und ODBC](#)

1.2 Enterprise Java Beans

Neben den bereits aus anderen Veranstaltungen bekannten Servlets und den davon abgeleiteten Java Server Pages bildet die Technik der *Enterprise Java Beans* (EJB) einen weiteren zentralen Baustein der Java 2 Enterprise Plattform. Als serverseitige Komponenten kommt den EJBs heute große Bedeutung in der Realisierung komplexer Anwendungen, insbesondere durch Umsetzung der sog. „Business Logik“, d.h. den nicht-interaktiven fachlichen Anwendungsteilen, zu.

Der Begriff der Enterprise Java Bean stützt sich auf dem historisch älteren der [Java Bean](#). Eine solche stellt eine abgeschlossene wiederverwendbare Softwarekomponente dar, die nach ihrer Erstellung über festgelegte Schnittstellen parametrisiert und manipuliert werden kann. Hierzu muß eine Bean eine festgelegte Interaktionsschnittstelle bieten, die durch die Java Bean Spezifikation definiert ist. Es handelt sich dabei um eine Reihe von Konventionen, der eine Bean gehorchen muß, jedoch um keine festgelegte API, die durch eine Komponente zu implementieren ist.

Der Begriff der Enterprise Java Bean greift diese inhaltliche Fundierung auf und präzisiert gleichzeitig die technische Umsetzung. So stellt eine Enterprise Java Bean eine Softwarekomponente dar, die in einer festgelegten Ausführungsumgebung, welche durch die EJB-Spezifikation festgelegte Dienste zur Nutzung durch die Beans anbieten kann. Eine solche Ausführungsumgebung wird als *Container* bezeichnet.

Ziel der Trennung in Komponente und Ausführungsumgebung ist die Zielsetzung die Enterprise Java Bean ausschließlich zur Umsetzung fachlicher Aufgaben heranzuziehen und alle infrastrukturellen Fragestellungen wie Betriebsmittelverwaltung, Persistenz oder Sicherheit durch die Ausführungsumgebung in gleicher Weise für alle Komponenten bereitzustellen.

Ein EJB-Container wird zumeist im Rahmen eines Application-Servers bereitgestellt.

Die gelegentlich anzutreffende Hervorhebung der anfänglich für Java Beans intendierten *visuellen Manipulationsmöglichkeit* trifft für Enterprise Java Beans nicht zu und hat sich für Java Beans auch nur begrenzte Bedeutung erlangt.

Spezifikationsgemäß können EJB-Container folgende Eigenschaften offerieren:

• Betriebsmittelverwaltung

Typischerweise verwaltet ein einziger EJB-Container gleichzeitig eine Reihe verschiedener Enterprise Java Beans. Zur Organisation und Aufrechterhaltung der Ausführbarkeit obliegt dem Container die Zuteilung von Betriebsmitteln wie Hauptspeicher, CPU-Zeit oder Netzwerkressourcen an die verwalteten EJB. Hierunter fällt insbesondere auch die Einlagerung, Instanziierung und Entfernung der EJBs selbst.

• Zustandsverwaltung

In praktischen Anwendungen ist oft die Nutzung zustandsbehafteter Kommunikation, die sich über

verschiedene Einzelinteraktionen erstreckt gewünscht. Die hierfür notwendigen technischen Voraussetzungen (Zustandsspeicherung, Korrelation der Einzelinteraktionen) werden durch den Container bereitgestellt.

- **Transaktionsverwaltung**

Erweiterung der Zustandsverwaltung. Zur Gewährleistung des benötigten Verhaltens müssen EJBs keine eigenen Implementierungen zur Verfügung stellen, sondern können vorhandene Dienste des Containers nutzen.

- **Sicherheit**

Die Sicherheit von EJBs kann durch Vergabe von Zugriffsrechten und Rollen auf Containerebene gesteuert werden.

- **Persistenz**

Der interne Zustand einer verwalteten EJB kann wahlfrei in persistiert und zu einem späteren Zeitpunkt wiederhergestellt werden.

- **Entfernter Zugriff**

Der Zugriff auf EJBs erfolgt mittels [Remote Method Invocation](#) und ist daher Lokationstransparent.

Neben den in der Aufzählung dargestellten Eigenschaften dürfen Container zusätzlich Weitere wahlfrei implementieren.

EJB-Typen

Grundsätzlich lassen sich alle EJBs drei Typen zuordnen: *Session Beans*, *Entity Beans* und *Message Driven Beans*. Während erstere hauptsächlich zur Abbildung von Abläufen eingesetzt werden, dienen Entity Beans der Abwicklung von Zugriffen auf Daten. Eine Sonderstellung nehmen die *Message Driven Beans* ein, die lediglich hinsichtlich ihres Kommunikationsverhaltens festgelegt sind.

Session Beans dienen der Abbildung von Abläufen im Rahmen der Programmierung der sog. Business Logik. Die Lebensdauer (d.h. Zeitspanne zwischen Erzeugung im und Entfernung aus dem Hauptspeicher) ist daher identisch mit der einer durch den Client erfolgenden Anfrage. Jede zu einem Zeitpunkt existierende Session Bean repräsentiert daher eine zugehörige Clientinstanz. Nach ihrer internen Ausgestaltung werden *stateless* und *statefull* Session Beans unterschieden. Während Erstere keinen über einen einzigen Aufruf hinausgehenden Zustand verwalten und daher seiteneffektfrei lediglich auf den durch den Aufruf übermittelten Daten operieren erhält das zustandserhaltende Pendant die Daten eines Aufrufs und kann diese auch in nachfolgenden Aufrufen verarbeiten.

Entity Beans sind programmiersprachliche Stellvertreter datenbankresidenter Objekte. Sie dienen dem erleichterten Zugriff auf persistent vorliegende Datenbestände. Ihre interne Realisierung ist eng mit der Technik relationaler Datenbankmanagement System verbunden. So werden Sie durch einen anwenderdefinierten [Primärschlüssel](#) dauerhaft identifiziert.

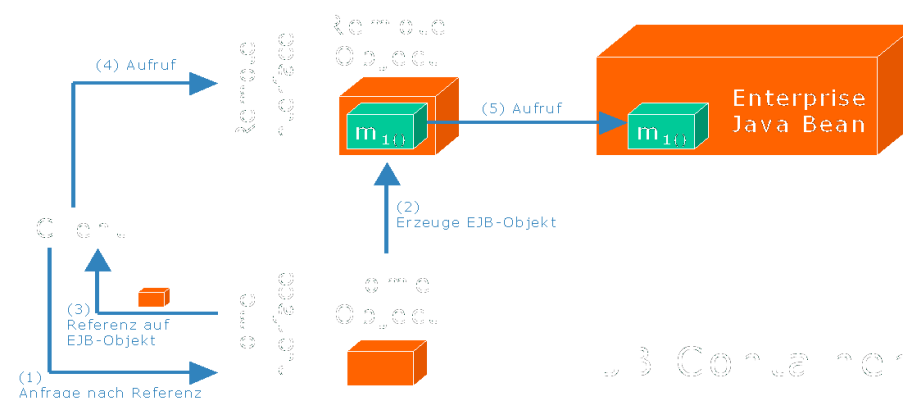
Message Driven Beans sind hinsichtlich ihres Kommunikationsverhaltens auf asynchrone Aufrufe beschränkt. Die Realisierung des eigentlichen Verhaltens wird durch eine Ausprägung eines der anderen Beantypen geboten.

Session Beans

Konzeptionell umfaßt jede EJB-Anwendung, die Session Beans einsetzt, die in [Abbildung 4](#) dargestellten Teile:

- Implementierung der EJB selbst.
- *Remote*-Schnittstelle zum Zugriff auf die durch die Bean publizierten Methoden.
- *Home*-Schnittstelle zur Ermittlung einer Referenz auf das Bean-Objekt.

Abbildung 4: Aufrufstruktur einer zustandslosen EJB



(click on image to enlarge!)

Gegenüber der Realisierung als RMI-Anwendung benötigt die Umsetzung als zustandslose Session Bean die Erstellung einer sog. „Home-Schnittstelle“ (*Home Interface*), welche die Operation `create` zur Instanziierung des serverseitigen EJB-Objekts bietet. Sie ist im Beispiel 23 dargestellt.

Beispiel 23: Home-Schnittstelle einer EJB

```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.CreateException;
(3)import javax.ejb.EJBHome;
(4)
(5)public interface SayHelloHome extends EJBHome {
(6)    public SayHello create() throws RemoteException, CreateException;
(7)}
```

[Download des Beispiels](#)

Die anwenderdefinierte Home-Schnittstelle erweitert die durch die Standard-API vorgegebene Schnittstelle `EJBHome`. Diese definiert Operationen zur Entfernung existierender EJB-Objekte aus dem Hauptspeicher (`remove`) sowie zur Ermittlung von Metadaten (`getEJBMetaData`) oder zum Erhalt eines netzwerkunabhängigen Verweises auf das EJB-Objekt (`getHomeHandle`).

Im Einzelnen sind dies die Operationen:

- `EJBMetaData getEJBMetaData()`
Liefert ein `EJBMetaData`-Objekt welches einzelne Eigenschaften einer EJB näher beschreibt. Hierzu zählen:
 - Klasse der Home-Schnittstelle
 - Klasse des Primärschlüssels (nur vorhanden sofern es sich um eine Entity Bean handelt)
 - Klasse der Remote-Schnittstelle
 - Boole'scher Wert, der angibt, ob es sich um eine Session Bean handelt
 - Boole'scher Wert, der angibt, ob es sich um eine zustandslose Session Bean handelt
- `HomeHandle getHomeHandle()`
Liefert ein Objekt des Typs `HomeHandle` zurück, welches eine netzwerkunabhängige Abstraktion des Verweises auf das Home-Objekt realisiert.
- `void remove (Handle h)`
Entfernt ein durch den Objektverweis (`Handle`) identifiziertes EJB-Objekt aus dem Hauptspeicher.
- `void remove (Object pk)`
Entfernt ein durch das übergebene Primärschlüsselobjekt identifiziertes EJB-Objekt aus dem Hauptspeicher.

Interessanterweise definiert die Schnittstelle zwar Operationen zur Ermittlung von Daten über bestehende Objekte und zur Entfernung dieser Objekte aus dem Hauptspeicher, nicht jedoch zu ihrer Erzeugung.

Dies liegt in der durch die Programmiersprache Java angestrebten statischen Typsicherheit begründet, die es nicht gestattet Operationen mit variablen Parameterlisten --- wie sie für die zum API-Erstellungszeitpunkt unbekannten spezifischen Initialisierungsparameter aller denkbaren EJBs benötigt würden --- zu versehen.

Aus diesem Grunde definiert die EJB-Spezifikation informell, daß ein diese Schnittstelle erweiternde eigene Home-Schnittstelle zusätzlich die Methode `create`, deren Signatur als Rückgabetyt den Typ der Remote-Schnittstelle vorsehen muß definiert. Zusätzlich enthält diese Operation die zur Initialisierung der Bean benötigten Parameter in ihrer Parameterliste.

Die im Beispiel 24 ist Remote-Schnittstelle dargestellt, deren Ausprägungen von Home-Objekten angesprochen werden:

Beispiel 24: Remote-Schnittstelle einer EJB

```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.EJBObject;
(3)
(4)public interface SayHello extends EJBObject {
(5)    public String sayHello(String name) throws RemoteException;
(6)}
```

[Download des Beispiels](#)

Schnittstellen dieses Typs enthalten ausschließlich die fachlichen Operationen, d.h. die Signaturen der Methoden, die später durch den Client benutzt werden.

Jede Remote-Schnittstelle erweitert zusätzlich die vorgegebene Schnittstelle `EJBObject`, welche,

ähnlich zur Home-Schnittstelle, einige Operationen zur Verwaltung eines EJB-Objektes vorgibt:

- `EJBHome getEJBHome()`
Liefert die Home-Schnittstelle einer EJB.
- `Handle getHandle()`
Liefert ein Objekt des Typs `HomeHandle` zurück, welches eine netzwerkunabhängige Abstraktion des Verweises auf das Home-Objekt realisiert.
- `Object getPrimaryKey()`
Liefert das Primärschlüsselobjekt einer Entity Bean.
- `boolean isIdentical(EJBObject eo)`
Prüft ob das übergebene EJB-Objekt dasselbe wie das Objekt ist auf dem die Methode ausgeführt wird.
- `void remove()`
Entfernt das EJB-Objekt aus dem Bean-Container.

Beispiel 25 zeigt die Implementierung der Bean selbst:

Beispiel 25: Realisierung einer Session Bean

```
(1)import java.rmi.RemoteException;
(2)import java.util.Date;
(3)import javax.ejb.EJBException;
(4)import javax.ejb.SessionBean;
(5)import javax.ejb.SessionContext;
(6)
(7)public class HelloWorldBean implements SessionBean {
(8)    public HelloWorldBean() {
(9)    }
(10)
(11)    public String sayHello(String name) {
(12)        return ("Hello " + name + " it is now " + new Date().toString());
(13)    }
(14)
(15)    public void setSessionContext(SessionContext arg0)
(16)        throws EJBException, RemoteException {
(17)    }
(18)    public void ejbRemove() throws EJBException, RemoteException {
(19)    }
(20)
(21)    public void ejbCreate() throws EJBException {
(22)    }
(23)
(24)    public void ejbActivate() throws EJBException, RemoteException {
(25)    }
(26)
(27)    public void ejbPassivate() throws EJBException, RemoteException {
(28)    }
(29)}
```



[Download des Beispiels](#)

Die programmiersprachliche Umsetzung der Bean enthält die Methoden der in der Remote-Schnittstelle bekanntgegebenen fachlichen Operationen. Zusätzlich muß ein Konstruktor expliziert werden, dessen Parameterliste mit den für die Operation `create` des Home-Interfaces gegebenen übereinstimmen.

Spezifikationsgemäß muß jede Session Bean die gleichnamige API-Schnittstelle implementieren. Diese definiert einige Operationen zur Behandlung unterschiedlicher Lebenszyklusstadien einer EJB. Hierunter fallen Methoden, die beim Erzeugen (`ejbCreate`), Entfernen (`ejbRemove`), bei der Passivierung (d.h. Auslagerung auf Hintergrundspeicher) (`ejbPassivate`) und dessen Reaktivierung (`ejbActivate`) eines EJB-Objekts durch die Ausführungsumgebung aufgerufen werden.

Bei der zwingend zu implementierenden Schnittstelle `SessionBean` handelt es sich nicht nur um eine Konvention um die Umsetzung der Lebenszyklusschnittstelle sicherzustellen, sondern auch um die Kategorisierung der Bean selbst. So stellt die im Beispiel verwandte Schnittstelle `SessionBean` neben `EntityBean` und `MessageDrivenBean` eine Spezialisierung der (operationslosen) Schnittstelle `EnterpriseBean` dar, deren „Implementierung“ durch eine Klasse lediglich zur Kennzeichnung dieser als EJB herangezogen wird.

Die genannten Spezialisierungen dieser Schnittstelle erfüllen daher sowohl den Zweck der Ausübung des Implementierungszwanges für die in ihnen aufgeführten Operationen als auch den der typisierenden Kennzeichnung.

Darüberhinaus ist `EnterpriseBean` als Spezialisierung der Standard-Schnittstelle [Serializable](#)

angelegt. In der Konsequenz muß jedes EJB-Objekt durch die Sprachmechanismen serialisierbar sein. Diese Eigenschaft wird insbesondere für die Passivierung und im Rahmen der Entity Beans genutzt.

Konzeptionell erinnert die Trennung in publizierte fachliche Schnittstelle (Remote-Schnittstelle) und deren technischer Umsetzung durch die EJB an die aus der Betrachtung des Remote Method Invocation Mechanismus bekannte Struktur.

Allerdings weicht die Umsetzung der Bean von der dort anzutreffenden Konvention ab die publizierte Schnittstelle selbst durch die realisierende Klasse zu implementieren. Dies liegt vor allem an der gegenüber RMI veränderten Struktur der publizierten Schnittstelle begründet. Während RMI für die Schnittstelle die Spezialisierung der operationslosen Standardschnittstelle [Remote](#) fordert die EJB-Spezifikation die Erweiterung der Schnittstelle [EJBObject](#), welche selbst die [oben dargestellten](#) Operationen definiert. Aus diesem Grunde würde die Aufnahme der Remote-Schnittstelle, obwohl konzeptionell durchaus zu rechtfertigen, in die Umsetzungsliste der EJB gleichzeitig die Implementierung von zumindest leeren Methodenrumpfen für die in EJBObject definierten Operationen notwendig werden lassen.

Abgesehen von dieser Ausnahme rekonstruiert das Verhältnis zwischen EJB und deren Remote-Schnittstelle die aus RMI bekannte Beziehung zwischen Schnittstelle und Umsetzung.

Die Nutzung einer durch eine Java Bean angebotenen Funktionalität erfolgt gemäß dem in [Abbildung 4](#) dargestellten Schema. Ein dies umsetzender Client ist in Beispiel 26 dargestellt.

Beispiel 26: Zugriff auf eine Session Bean

```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.CreateException;
(3)import javax.naming.Context;
(4)import javax.naming.InitialContext;
(5)import javax.naming.NamingException;
(6)import javax.rmi.PortableRemoteObject;
(7)
(8)public class CallHelloWorldBean {
(9)    public static void main(String[] args) {
(10)        try {
(11)            Context initial = new InitialContext();
(12)            Object objRef = initial.lookup("helloBean");
(13)
(14)            SayHelloHome home =
(15)                (SayHelloHome) PortableRemoteObject.narrow(
(16)                    objRef,
(17)                    SayHelloHome.class);
(18)            SayHello sh = home.create();
(19)
(20)            System.out.println(sh.sayHello("Mario"));
(21)
(22)        } catch (NamingException ne) {
(23)            ne.printStackTrace();
(24)        } catch (RemoteException re) {
(25)            re.printStackTrace();
(26)        } catch (CreateException ce) {
(27)            ce.printStackTrace();
(28)        }
(29)    }
(30)}
```



[Download des Beispiels](#)

Zunächst ermittelt der Client unter Nutzung der JNDI-API eine Referenz auf die EJB. Dies geschieht durch Anfrage (lookup) an den JNDI-Dienst unter Übergabe des bekannten Klarnamens (*helloBean*). Die erhaltene generische Referenz wird durch Aufruf der statischen Methode [narrow](#) der Klasse [PortableRemoteObject](#) typischer in eine Ausprägung der Home-Schnittstelle konvertiert. Der Aufruf der in dieser Schnittstelle durch den Anwender definierten create-Methode sorgt für die serverseitige Instanziierung der EJB, die als Ausprägung der Remote-Schnittstelle geliefert wird. Tatsächlich wird nicht das EJB-Objekt selbst durch den Methodenaufwurf retourniert, sondern lediglich ein netzwerktransparenter Verweis darauf, der jedoch clientseitig einer lokalen Objektreferenz gleichgestellt verwendet werden kann. Ferner wird serverseitig zur Kommunikation mit der EJB ein Home-Objekt erzeugt, welchem eine Stellvertreterrolle für den anfragenden Client zukommt. Der Aufruf der durch die EJB zur Verfügung gestellten Methode erfolgt identisch zu dem einer Lokalen.

Entity Beans

Die zweite zentrale Klasse von Enterprise Java Beans bilden die zur serverseitigen Persistierung von Objekten dienenden *Entity Beans*.

Sie kapseln Datenbankinhalte durch Objekte, die gemäß der EJB-Spezifikation instanzierbar und zugreifbar sind. Die Verwaltung der gekapselten Dateninhalte erfolgt durch ein frei festlegbares Datenbankmanagementsystem, die der Objekte selbst durch den EJB-Container.

Ziel dieser Technik ist es die Komplexität der Persistenzlogik für den Verwender der bereitgestellten Bean vollkommen transparent zu gestalten und serverseitig zu realisieren.

Die Familie der Entity Beans selbst zerfällt in zwei Untertypen, welche sich entlang des Realisierungspunktes der Persistenzlogik separieren: *Bean Managed Persistence*, der Bean obliegt die Umsetzung der Persistenzlogik, und *Container Managed Persistence*, hierbei wird die Persistenzlogik durch den EJB-Container realisiert.

Das nachfolgende Beispiel zeigt die Umsetzung einer Entity Bean mit Bean Managed Persistence. Es kapselt die Verwaltung und den Zugriff auf Objekte, die Personen beschreiben. Jedes Personen-Objekt enthält Daten zu Name, Geburtsdatum und Wohnstraße. Der Name dient als eindeutige Identifikation und daher datenbankseitig als Primärschlüssel. Die notwendige Datenbanktabelle wurde erzeugt durch den SQL-Ausdruck:

```
CREATE TABLE PERSON( Name VARCHAR(20) PRIMARY KEY, Birthdate DATE, Street VARCHAR(30));
```

Wie bereits für die Realisierung von Session Beans eingeführt, werden auch zur Publikation der extern zugänglichen Schnittstellen Home und Remote Interfaces benötigt.

Struktur und Aufbau der Home-Schnittstelle ähnelt konzeptionell der für Session Beans eingeführten. Dieser Schnittstellentyp dient auch für Entity Beans zur Aufnahme der Verwaltungsoperationen zur Erzeugung (create) und zur Suche existierender EJBs (findByPrimaryKey).

Beispiel 27 zeigt die Home-Schnittstelle des Beispiels.

Beispiel 27: Home-Schnittstelle einer Entity Bean



```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.CreateException;
(3)import javax.ejb.EJBHome;
(4)import javax.ejb.FinderException;
(5)
(6)public interface PersonHome extends EJBHome {
(7)    public Person create(int year, int month, int day, String name, String
street) throws CreateException, RemoteException;
(8)    public Person findByPrimaryKey(String name) throws FinderException,
RemoteException;
(9)}
```

[Download des Beispiels](#)

Die Home-Schnittstelle zeigt die create-Operation zur Erzeugung einer neuen EJB-Instanz. Ihre Übergabeparameter dienen zur Konstruktion des neuen Objekts und werden durch die Bean-Implementierung interpretiert.

Ferner enthält die Schnittstelle mit findByPrimaryKey eine Operation deren Implementierung eine Entity-Bean anhand ihres Primärschlüssels identifiziert und liefert. Aus diesem Grunde erhält die Methode den zu suchenden Wert von Typ des Primärschlüssels übergeben.

Hinsichtlich der verwendeten Typen zeigt sich bereits hier, daß eine Abbildung der durch die Programmiersprache Java bereitgestellten Typen auf die des eingesetzten Persistenzsystems stattfinden muß. In der im Beispiel gewählten Ausführungsform der durch die EJB selbst verwalteten Persistenz muß diese Abbildung manuell durch den Programmierer bereitgestellt werden.

Die Remote-Schnittstelle gibt die für Nutzer der Bean zugänglichen Geschäftsfunktionen wieder. Daher enthält dieser Schnittstellentyp lediglich Operationen zum Zugriff auf die verwalteten Daten, nicht jedoch zur technischen Verwaltung und Interaktion mit dem Bean-Container.

Per Konvention muß diese Schnittstelle als Spezialisierung der Schnittstelle [EJBObject](#) definiert sein. Diese Standardschnittstelle definiert allgemeine Interaktionsformen, wie Löschen (remove), Vergleich (isIdentical) und Ermittlung des Primärschlüsselwertes (getPrimaryKey) die für alle Entity Bean Objekte gleichermaßen benötigt werden.

[isIdentical](#) liefert den Vergleich zweier serverseitiger EJB-Objekte und ermittelt so, ob zwei Java-

Objektreferenzen auf dasselbe Datenbankobjekt zugreifen.

Mittels [getPrimaryKey](#) ermittelt den Wert des Primärschlüssels eines gegebenen EJB-Objekts aus der Datenbank.

Zur Lösung von datenbankresidenten Objekten wird [remove](#) eingesetzt. Der Aufruf dieser Methode entfernt ausschließlich die durch die EJB repräsentierten Datenbanktupel, die programmiersprachliche Objektrepräsentation bleibt jedoch über die gesamte Laufzeit (sofern nicht durch Gültigkeitsbereiche oder explizite NULL-Setzung explizit anders gehandhabt) intakt.

Beispiel 28: Remote-Schnittstelle einer Entity Bean



```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.EJBObject;
(3)
(4)public interface Person extends EJBObject {
(5)    public int getAge() throws RemoteException;
(6)    public String getStreet() throws RemoteException;
(7)    public void setStreet(String street) throws RemoteException;
(8)}
```

[Download des Beispiels](#)

Beispiel 29 zeigt den vollständigen Code der Bean. Sie implementiert mit [EntityBean](#) die Standardschnittstelle alle Entity Beans, welche als Spezialisierung der ausschließlich markierenden Schnittstelle [EnterpriseBean](#) die notwendigen Basisoperationen zur Abwicklung der persistenten Speicherung.

Im Falle einer *Bean Managed Persistence* enthalten die Methoden der durch die Schnittstelle definierten und der zusätzlich im Rahmen der Spezifikation textuell definierten Operationen die notwendigen Aufrufe zur Ablage eines Objekts in der Datenbank und zu seiner späteren Extraktion daraus.

Im Einzelnen sind dies die Operationen:

Tabelle 1: Persistenzoperationen einer Entity Bean



Operation	Semantik	Zugehörige SQL-Anweisung
<code>ejbCreate</code>	Wird nach dem Erzeugen eines Java-Objektes aufgerufen um dieses in der Datenbank abzulegen. Diese Operation ist nicht Bestandteil der Schnittstelle, da ihre Parameter, die den Übergabeparametern des Objektkonstruktors entsprechen, zum Schnittstellenerzeugungszeitpunkt nicht feststehen.	INSERT
<code>ejbFindByPrimaryKey</code>	Liefert den Wert des Primärschlüssels zurück, sofern ein Datenbankeintrag existiert, der durch diesen Primärschlüssel identifiziert wird. Diese Operation ist nicht Bestandteil der Schnittstelle, da ihre Parameter, die in Typ, Name und Reihenfolge der Zusammensetzung des Primärschlüssels entsprechen, zum Schnittstellenerzeugungszeitpunkt nicht feststehen. Diese Methode wird nicht durch den Anwender direkt aufgerufen, sondern stattdessen auf einer Ausprägung der Home-Schnittstelle das dort zur Verfügung stehende Analogon <code>findByPrimaryKey</code> , welches das durch den Primärschlüssel identifizierte EJB-Objekt zurückliefert. Diese Methode greift intern auf ausschließlich den Schlüssel liefernde Methode der Bean zu.	SELECT
ejbRemove	Entfernt das EJB-Objekt aus der Datenbank.	DELETE

ejbStore	Synchronisiert das Java-Objekt mit dem EJB-Objekt und aktualisiert so die Datenbankinhalte. Diese Methode wird nach jedem Zugriff mittels einer in der Remote-Schnittstelle aufgeführten Operation auf das EJB-Objekt ausgeführt.	UPDATE

Beispiel 29: Eine Entity Bean

```

(1)import java.rmi.RemoteException;
(2)import java.sql.Connection;
(3)import java.sql.DriverManager;
(4)import java.sql.ResultSet;
(5)import java.sql.SQLException;
(6)import java.sql.Statement;
(7)import java.util.Calendar;
(8)import java.util.GregorianCalendar;
(9)
(10)import javax.ejb.CreateException;
(11)import javax.ejb.EJBException;
(12)import javax.ejb.EntityBean;
(13)import javax.ejb.EntityContext;
(14)import javax.ejb.FinderException;
(15)import javax.ejb.RemoveException;
(16)
(17)public class PersonBean implements EntityBean {
(18)    //persistent fields
(19)    private String name;
(20)    private GregorianCalendar birthdate;
(21)    private String street;
(22)
(23)    public PersonBean() {
(24)    }
(25)
(26)    //methods which are part of the remote interface
(27)    public int getAge() {
(28)        return (
(29)            new GregorianCalendar().get(Calendar.YEAR)
(30)            - birthdate.get(Calendar.YEAR));
(31)    }
(32)    public String getStreet() {
(33)        return street;
(34)    }
(35)    public void setStreet(String street) throws RemoteException {
(36)        if (street.length() <= 30) {
(37)            this.street = street;
(38)        } else {
(39)            throw new RemoteException("cannot update street");
(40)        }
(41)    }
(42)    // -----
(43)    public String ejbCreate(
(44)        int year,
(45)        int month,
(46)        int day,
(47)        String name,
(48)        String street)
(49)        throws CreateException {
(50)        if (year > 1900
(51)            && month >= 1
(52)            && month <= 12
(53)            && day >= 1
(54)            && day <= 31
(55)            && name.length() <= 20
(56)            && street.length() <= 30) {
(57)
(58)            this.name = name;
(59)            this.birthdate = new GregorianCalendar(year, month, day);
(60)            this.street = street;
(61)            Statement stmt = getStatement();

```

```

(62)         String s =
(63)             new String(
(64)                 "INSERT INTO PERSON VALUES ('"
(65)                     + name
(66)                     + "',''"
(67)                     + birthdate.get(Calendar.YEAR)
(68)                     + "-"
(69)                     + birthdate.get(Calendar.MONTH)
(70)                     + "-"
(71)                     + birthdate.get(Calendar.DATE)
(72)                     + "',''"
(73)                     + "'"
(74)                     + street
(75)                     + "'"
(76)                     + "');"");
(77)         try {
(78)             stmt.executeUpdate(s);
(79)         } catch (SQLException e) {
(80)             e.printStackTrace();
(81)         }
(82)     } else {
(83)         throw new CreateException("Invalid values supplied");
(84)     }
(85)     return name;
(86) }
(87) public void ejbPostCreate(
(88)     int year,
(89)     int month,
(90)     int day,
(91)     String name,
(92)     String street) {
(93) }
(94) // -----
(95) public int ejbHomeGetAge() {
(96)     return 0;
(97) }
(98)
(99) public String ejbHomeGetStreet() {
(100)     return new String();
(101) }
(102)
(103) public void ejbHomeSetStreet(String street) {
(104) }
(105)
(106) public Statement getStatement() {
(107)     try {
(108)         Class.forName("com.mysql.jdbc.Driver");
(109)     } catch (ClassNotFoundException e) {
(110)         e.printStackTrace();
(111)     }
(112)     Connection con = null;
(113)     try {
(114)         con =
(115)             (Connection) DriverManager.getConnection(
(116)                 "jdbc:mysql://10.0.0.1/Address/",
(117)                 "mario",
(118)                 "thePassword");
(119)     } catch (SQLException e1) {
(120)         e1.printStackTrace();
(121)     }
(122)     try {
(123)         return ((Statement) con.createStatement());
(124)     } catch (SQLException e2) {
(125)         e2.printStackTrace();
(126)     }
(127)     return null; //never gets here
(128) }
(129)
(130) // -----
(131) public void setEntityContext(EntityContext ectx)
(132)     throws EJBException, RemoteException {
(133) }

```




```

(134)
(135) public void unsetEntityContext() throws EJBException, RemoteException {
(136) }
(137)
(138) public void ejbRemove()
(139)     throws RemoveException, EJBException, RemoteException {
(140)     Statement stmt = getStatement();
(141)     String s = new String("DELETE FROM PERSON WHERE Name='" + name +
";");
(142)     try {
(143)         stmt.executeUpdate(s);
(144)     } catch (SQLException e) {
(145)         e.printStackTrace();
(146)     }
(147) }
(148)
(149) public void ejbActivate() throws EJBException, RemoteException {
(150) }
(151)
(152) public void ejbPassivate() throws EJBException, RemoteException {
(153) }
(154)
(155) public void ejbLoad() throws EJBException, RemoteException {
(156) }
(157)
(158) public void ejbStore() throws EJBException, RemoteException {
(159)     Statement stmt = getStatement();
(160)     String s =
(161)         new String(
(162)             "UPDATE PERSON SET Name='"
(163)                 + name
(164)                 + "', BDATE='"
(165)                 + birthdate.get(Calendar.YEAR)
(166)                 + "-"
(167)                 + birthdate.get(Calendar.MONTH)
(168)                 + "-"
(169)                 + birthdate.get(Calendar.DATE)
(170)                 + "', Street = '"
(171)                 + street
(172)                 + "' WHERE Name = '"
(173)                 + name
(174)                 + "';";
(175)     try {
(176)         stmt.executeUpdate(s);
(177)     } catch (SQLException e) {
(178)         e.printStackTrace();
(179)     }
(180) }
(181)
(182) public String ejbFindByPrimaryKey(String name) throws FinderException {
(183)     Statement stmt = getStatement();
(184)     String s =
(185)         new String("SELECT Name FROM PERSON WHERE Name='" + name +
";");
(186)     try {
(187)         ResultSet rs = stmt.executeQuery(s);
(188)         rs.first();
(189)         return rs.getString("Name");
(190)     } catch (SQLException e) {
(191)         e.printStackTrace();
(192)     }
(193)     return null; //never gets here
(194) }
(195)}

```

[Download des Beispiels](#)

Zusätzlich enthält die Bean des Beispiels mit `getAge` eine zwar in der Remote-Schnittstelle veröffentlichte Operation, die keinen direkt abgespeicherten Wert liefert, sondern diesen dynamisch zur Ausführungszeit anhand der verfügbaren Daten berechnet.

Alle anderen in der Remote-Schnittstelle aufgeführten Operationen (etwa: `getStreet`, `setStreet`) modifizieren lediglich, den durch die Attribute repräsentierten Java-Objektzustand und greifen nicht direkt auf die Datenbank zu.

Innerhalb der Datenbankzugreifenden Methoden muß durch den Anwender die Abbildung der Java-Datentypen auf die des verwendeten Datenbankmanagementsystems erfolgen. Die mit dem Präfix `ejb` versehenen Methoden zeigen dies für die lesenden und schreibenden DB-Zugriffe. So kann die im Beispiel für `name` und `street` verwendete Java-Repräsentation `String` vergleichsweise leicht in den SQL-Typ `VARCHAR` abgebildet werden sofern durch alle Methoden, die Datenbankinhalte schreiben sicherstellen, daß nur zum Datenbankschema konforme Werte eingefügt werden. Die Beispielimplementierung zeigt dies exemplarisch anhand der Methoden `ejbCreate` und `setStreet`. Für programmiersprachliche Typen, die nicht direkt in DB-Typen abbildbar sind muß im Falle der Bean Managed Persistence der Bean-Entwickler selbst Sorge für die adäquate Abbildung tragen. Das Beispiel illustriert dies anhand des Java-Datumstyps [GregorianCalendar](#), der manuell in die durch das DBMS erwartete [ISO 8601](#)-konforme Darstellung zu überführen ist.

Einige der möglichen Interaktionen mit der Bean zeigt der Code des Clients aus Beispiel 30:

Beispiel 30: Client der auf eine Entity Bean zugreift

```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.CreateException;
(3)import javax.ejb.FinderException;
(4)import javax.ejb.RemoveException;
(5)import javax.naming.Context;
(6)import javax.naming.InitialContext;
(7)import javax.naming.NamingException;
(8)import javax.rmi.PortableRemoteObject;
(9)
(10)public class CallPersonBean {
(11)    public static void main(String[] args) {
(12)        try {
(13)            Context initial = new InitialContext();
(14)            Object objRef = initial.lookup("personBean");
(15)
(16)            PersonHome home =
(17)                (PersonHome) PortableRemoteObject.narrow(
(18)                    objRef,
(19)                    PersonHome.class);
(20)            Person p1 = home.create(1911, 1, 1, "Alice", "streetA");
(21)            Person p2 = home.create(1922, 2, 2, "Bob", "streetB");
(22)            System.out.println("Alice's Age: " + p1.getAge());
(23)
(24)            System.out.println("Alice's primary key: " + p1.getPrimaryKey
(25)                ());
(26)            p1.remove();
(27)
(28)            Person p3 = home.findByPrimaryKey("Bob");
(29)            System.out.println(
(30)                "Bob's modified street (before modification): "
(31)                + p3.getStreet());
(32)            p3.setStreet("streetC");
(33)            System.out.println(
(34)                "Bob's modified street (after modification): "
(35)                + p3.getStreet());
(36)            System.out.println("also the other reference: " + p2.
(37)                getStreet());
(38)
(39)            System.out.println("Are both references the same Java
(40)                object: " + (p2==p3));
(41)            System.out.println("Are both references the same EJBObject
(42)                (calls isIdentical): " + p2.isIdentical(p3));
(43)
(44)        } catch (NamingException ne) {
(45)            ne.printStackTrace();
(46)        } catch (RemoteException re) {
(47)            re.printStackTrace();
(48)        } catch (CreateException ce) {
(49)            ce.printStackTrace();
(50)        } catch (RemoveException e) {
(51)            e.printStackTrace();
(52)        } catch (FinderException e) {
```



```

(49)                e.printStackTrace();
(50)            }
(51)    }
(52)}}

```

[Download des Beispiels](#)

Der Client ermittelt zunächst per JNDI eine Referenz auf die Bean, welche unter dem Namen *personBean* im Verzeichnisdienst registriert ist.

Die erhaltene generische Referenz wird durch Aufruf der statischen Methode [narrow](#) der Klasse [PortableRemoteObject](#) typsicher in eine Ausprägung der Home-Schnittstelle (*PersonHome*) konvertiert.

Der Aufruf der in dieser Schnittstelle durch den Anwender definierten *create*-Methode sorgt für die serverseitige Instanziierung der EJB, die als Ausprägung der Remote-Schnittstelle geliefert wird. Tatsächlich wird nicht das EJB-Objekt selbst durch den Methodenaufruf retourniert, sondern lediglich ein netzwerktransparenter Verweis darauf, der jedoch clientseitig einer lokalen Objektreferenz gleichgestellt verwendet werden kann.

Ferner wird serverseitig zur Kommunikation mit der EJB ein Home-Objekt erzeugt, welchem eine Stellvertreterrolle für den anfragenden Client zukommt.

Der Aufruf der durch die EJB zur Verfügung gestellten Methode erfolgt identisch zu dem einer Lokalen.

So dient der Aufruf der Methode *create* zur Erzeugung von serverseitig instanziiert und transparent persistierten EJB-Objekten sowie den lokalen Java-(Stellvertreter-)Objekten für den Zugriff darauf.

Der Aufruf von *getAge* zeigt die Nutzung einer in der Remote-Schnittstelle veröffentlichten Zugriffsmethode. Mit *getPrimaryKey* wird die, in der durch die Remote-Schnittstelle erweiterten Schnittstelle *EJBObject* angesiedelte, Operation zur Ermittlung des Primärschlüsselwertes eines EJB-Objektes aufgerufen.

Die Methode *remove* stellt dagegen eine durch die Home-Schnittstelle definierte Operation dar.

Durch den Aufruf dieser Methode auf dem durch *p1* referenzierten Objekt wird werden durch durch Ausführung der Beanmethode *ejbRemove* die die Bean serverseitig repräsentierenden Datenbankseinträge entfernt sowie der durch die Bean belegte Speicherbereich als frei markiert. Alle Versuche nach Aufruf dieser Methode auf der clientseitigen hauptspeicherrepräsentation Wertänderungen durchzuführen führen daher zu einem Fehler.

Die Ermittlung von Referenzen auf existierende EJB-Objekte erfolgt durch die in der Remote-Schnittstelle definierte Methode *findByPrimaryKey*. Der EJB-Container stellt sicher, daß verschiedene Referenzen auf dasselbe EJB-Objekt synchronisiert in die Datenbank abgebildet werden, so daß keine Inkonsistenzen entstehen.

Für den Betrieb einer Enterprise Java Bean ist neben den bisher betrachteten Schnittstellen-Komponenten und der Realisierung der Bean selbst auch ein als *Deployment Descriptor* bezeichnetes XML-Konfigurationsfile notwendig, welches verschiedene Einstellungsdaten sowie die Schnittstellendaten enthält.

Beispiel 31 zeigt ein Beispiel hierfür:

Beispiel 31: Deployment Descriptor der Entity Bean

```

(1)<?xml version="1.0" encoding="UTF-8"?>
(2)
(3)<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN" 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
(4)
(5)<ejb-jar>
(6)  <display-name>Ejb1</display-name>
(7)  <enterprise-beans>
(8)    <entity>
(9)      <display-name>PersonBean</display-name>
(10)     <ejb-name>PersonBean</ejb-name>
(11)     <home>PersonHome</home>
(12)     <remote>Person</remote>
(13)     <ejb-class>PersonBean</ejb-class>
(14)     <persistence-type>Bean</persistence-type>
(15)     <prim-key-class>java.lang.String</prim-key-class>
(16)     <reentrant>False</reentrant>
(17)     <security-identity>
(18)       <description></description>
(19)     </security-identity>
(20)   </entity>
(21) </enterprise-beans>
(22)</ejb-jar>

```

```

(22) </enterprise-beans>
(23) <assembly-descriptor>
(24)   <method-permission>
(25)     <unchecked />
(26)     <method>
(27)       <ejb-name>PersonBean</ejb-name>
(28)       <method-intf>Remote</method-intf>
(29)       <method-name>getAge</method-name>
(30)       <method-params />
(31)     </method>
(32)     <method>
(33)       <ejb-name>PersonBean</ejb-name>
(34)       <method-intf>Home</method-intf>
(35)       <method-name>remove</method-name>
(36)       <method-params>
(37)         <method-param>javax.ejb.Handle</method-param>
(38)       </method-params>
(39)     </method>
(40)     <method>
(41)       <ejb-name>PersonBean</ejb-name>
(42)       <method-intf>Home</method-intf>
(43)       <method-name>getHomeHandle</method-name>
(44)       <method-params />
(45)     </method>
(46)     <method>
(47)       <ejb-name>PersonBean</ejb-name>
(48)       <method-intf>Remote</method-intf>
(49)       <method-name>isIdentical</method-name>
(50)       <method-params>
(51)         <method-param>javax.ejb.EJBObject</method-param>
(52)       </method-params>
(53)     </method>
(54)     <method>
(55)       <ejb-name>PersonBean</ejb-name>
(56)       <method-intf>Home</method-intf>
(57)       <method-name>remove</method-name>
(58)       <method-params>
(59)         <method-param>java.lang.Object</method-param>
(60)       </method-params>
(61)     </method>
(62)     <method>
(63)       <ejb-name>PersonBean</ejb-name>
(64)       <method-intf>Remote</method-intf>
(65)       <method-name>getHandle</method-name>
(66)       <method-params />
(67)     </method>
(68)     <method>
(69)       <ejb-name>PersonBean</ejb-name>
(70)       <method-intf>Home</method-intf>
(71)       <method-name>findByPrimaryKey</method-name>
(72)       <method-params>
(73)         <method-param>java.lang.String</method-param>
(74)       </method-params>
(75)     </method>
(76)     <method>
(77)       <ejb-name>PersonBean</ejb-name>
(78)       <method-intf>Home</method-intf>
(79)       <method-name>getEJBMetaData</method-name>
(80)       <method-params />
(81)     </method>
(82)     <method>
(83)       <ejb-name>PersonBean</ejb-name>
(84)       <method-intf>Remote</method-intf>
(85)       <method-name>getPrimaryKey</method-name>
(86)       <method-params />
(87)     </method>
(88)     <method>
(89)       <ejb-name>PersonBean</ejb-name>
(90)       <method-intf>Remote</method-intf>
(91)       <method-name>remove</method-name>
(92)       <method-params />
(93)     </method>

```



```

(94)      <method>
(95)          <ejb-name>PersonBean</ejb-name>
(96)          <method-intf>Remote</method-intf>
(97)          <method-name>getEJBHome</method-name>
(98)          <method-params />
(99)      </method>
(100) </method-permission>
(101) <container-transaction>
(102)     <method>
(103)         <ejb-name>PersonBean</ejb-name>
(104)         <method-intf>Home</method-intf>
(105)         <method-name>create</method-name>
(106)         <method-params>
(107)             <method-param>int</method-param>
(108)             <method-param>int</method-param>
(109)             <method-param>int</method-param>
(110)             <method-param>java.lang.String</method-param>
(111)             <method-param>java.lang.String</method-param>
(112)         </method-params>
(113)     </method>
(114)     <trans-attribute>Never</trans-attribute>
(115) </container-transaction>
(116) <container-transaction>
(117)     <method>
(118)         <ejb-name>PersonBean</ejb-name>
(119)         <method-intf>Remote</method-intf>
(120)         <method-name>setStreet</method-name>
(121)         <method-params>
(122)             <method-param>java.lang.String</method-param>
(123)         </method-params>
(124)     </method>
(125)     <trans-attribute>Never</trans-attribute>
(126) </container-transaction>
(127) <container-transaction>
(128)     <method>
(129)         <ejb-name>PersonBean</ejb-name>
(130)         <method-intf>Remote</method-intf>
(131)         <method-name>getStreet</method-name>
(132)         <method-params />
(133)     </method>
(134)     <trans-attribute>Never</trans-attribute>
(135) </container-transaction>
(136) <container-transaction>
(137)     <method>
(138)         <ejb-name>PersonBean</ejb-name>
(139)         <method-intf>Home</method-intf>
(140)         <method-name>remove</method-name>
(141)         <method-params>
(142)             <method-param>javax.ejb.Handle</method-param>
(143)         </method-params>
(144)     </method>
(145)     <trans-attribute>Never</trans-attribute>
(146) </container-transaction>
(147) <container-transaction>
(148)     <method>
(149)         <ejb-name>PersonBean</ejb-name>
(150)         <method-intf>Home</method-intf>
(151)         <method-name>remove</method-name>
(152)         <method-params>
(153)             <method-param>java.lang.Object</method-param>
(154)         </method-params>
(155)     </method>
(156)     <trans-attribute>Never</trans-attribute>
(157) </container-transaction>
(158) <container-transaction>
(159)     <method>
(160)         <ejb-name>PersonBean</ejb-name>
(161)         <method-intf>Remote</method-intf>
(162)         <method-name>remove</method-name>
(163)         <method-params />
(164)     </method>
(165)     <trans-attribute>Never</trans-attribute>

```

```

(166)    </container-transaction>
(167)    <container-transaction>
(168)        <method>
(169)            <ejb-name>PersonBean</ejb-name>
(170)            <method-intf>Remote</method-intf>
(171)            <method-name>getAge</method-name>
(172)            <method-params />
(173)        </method>
(174)        <trans-attribute>Never</trans-attribute>
(175)    </container-transaction>
(176)    <container-transaction>
(177)        <method>
(178)            <ejb-name>PersonBean</ejb-name>
(179)            <method-intf>Home</method-intf>
(180)            <method-name>findByPrimaryKey</method-name>
(181)            <method-params>
(182)                <method-param>java.lang.String</method-param>
(183)            </method-params>
(184)        </method>
(185)        <trans-attribute>Never</trans-attribute>
(186)    </container-transaction>
(187) </assembly-descriptor>
(188)</ejb-jar>
(189)

```

[Download des Beispiels](#)

1.3 Java Data Objects

Grundidee

Hintergrund des Ansatzes der *Java Data Objects* (JDO) ist es, die bestehenden Schnittstellenmechanismen dahingehend weiterzuentwickeln, daß die Persistenz von Objekten und Objektgraphen für den Programmierer vollständig transparent durch Komponenten der Laufzeitumgebung zur Verfügung gestellt werden.

Gleichzeitig etabliert JDO eine Abstraktion der verschiedenen Speicherungsmöglichkeiten und erlaubt es beispielsweise die dateibasierte Ablage innerhalb des Programmes identisch zur Objektspeicherung in einem Datenbankmanagementsystem zu handhaben. Auf dieser Basis läßt sich im Bedarfsfalle den Persistenzdienstleister auszutauschen ohne Änderungen am Programmcode zu erfordern.

Plakativ wird der Ansatz daher, in Anlehnung an die Zielsetzung der Programmiersprache Java des *write once -- run anywhere*, als *write once -- store anywhere* charakterisiert.

Technik

Um die weitestgehend transparente Handhabung der Objektpersistenz zu gewährleisten bedient sich JDO eines Ansatzes der über das alleinige Angebot einer Programmierschnittstelle hinausreicht. Die Zielsetzung der möglichst einfach handzuhabenden Interaktion mit den generischen Persistenzmechanismen läßt sich zwar durch das Angebot von durch den Programmierer zu implementierenden Schnittstellen und Persistenzklassen erreichen, jedoch ist der Einsatz signifikant komplexer als der bestehenden Persistenzschnittstellen. Darüberhinaus konterkariert der Zwang bei der Programmerstellung vorgegebene Schnittstellen zu berücksichtigen die Zielsetzung weitestgehender Transparenz der angebotenen Speichermechanismen.

Daher führt JDO die Technik der sog. *Bytecodeanreicherung* (engl. *bytecode enhancing*) ein. Hierbei wird durch eine Programmkomponente vorübersetzer Bytecode so abgeändert, daß die notwendigen Persistenzanweisungen in den bereits erzeugten ausführbaren Bytecode eingewoben werden.

Die benötigte Übersetzerkomponente wird durch die jeweilige JDO-Implementierung zur Verfügung gestellt und muß durch den Programmierer im Bedarfsfalle lediglich geeignet parametrisiert werden.

Im Falle der Referenzimplementierung müssen daher alle Klassen, die Objekte ausprägen, welche persistiert werden sollen, mit dem Werkzeug entsprechend nachbearbeitet werden. Der notwendige Aufruf hat folgende Struktur: `java com.sun.jdori.enhancer.Main -d enhanced de/jeckle/jdotest/Employee.class de/jeckle/jdotest/Employee.jdo`.

Dieser Aufruf reicht die bereits übersetzte Klasse `Employee` innerhalb der Pakethierarchie `de`.

jeckle.jdotest um Persistenzdaten an und legt das Ergebnis innerhalb des Dateisystemkatalogs de/jeckle/jdotest ab. Zur Anreicherung wird die Konfigurationsdatei Employee.jdo herangezogen, die im selben Pfad abgelegt ist wie die Quellcodedatei.

Alternativ zu diesem Ansatz steht auch die Möglichkeit zur Verfügung die benötigten Anweisungen bereits im Quellcode vorzusehen um so dasselbe Resultat zu erzielen, welches durch den Anreicherungsprozeß erzeugt wird. Diese Vorgehensweise hat jedoch wegen der damit verbundenen Aufwände kaum praktische Bedeutung erlangt und wird daher im folgenden nicht vertieft betrachtet.

Die Beispiele dieses Kapitels basieren auf der kostenfrei verfügbaren JDO-Referenzimplementierung von SUN. Diese beschränkt zwar die unterstützten Persistenzmechanismen auf ausschließlich dateibasierte Speicherung und sieht keine Ablage in Datenbankmanagementsystemen vor. Konzeptionell und programmierseitig ist die Interaktion mit dieser Implementierung jedoch identisch zu kommerziell verfügbaren Lösungen und können daher ohne weiteres auf diese und damit beliebige Persistenzdienstleister übertragen werden.

Konfiguration des Persistenzdienstleisters

Die Abbildung der in der Programmiersprache formulierten Interaktionen auf den konkreten physischen Persistenzdienstleister erfolgt sinnvollerweise an einer für alle JDO-nutzenden Applikationen zugänglichen Stelle im Rahmen einer Property-Datei. Die Inhalte dieser Datei unterscheiden naturgemäß bei den verschiedenen JDO-Herstellern und inhärent mit dem gewählten Persistenztyp. So benötigt die dateibasierte Objektablage offenkundig andere Festlegungen als der Zugriff auf ein relationales Datenbankmanagementsystem. Beispiel 32 zeigt die notwendigen Einstellung zur Konfiguration der dateibasierten Speicherung mit der SUN-Referenzimplementierung. Dort wird mit der PersistenceManagerFactoryClass diejenige Klasse innerhalb des JDO-Rahmenwerkes benannt, welche dem Programmierer die Persistenzdienste zur Verfügung stellt. ConnectionURL bildet das Bindeglied der Abbildung auf die physische Datei und benennt daher den Speicherort aller persistierten Objekte. Die zusätzlichen Angaben dienen der Authentisierung und Zugriffssteuerung beim Zugriff auf die erstellte Datei.

Beispiel 32: Konfiguration einer JDO-Implementierung



```
(1)javax.jdo.PersistenceManagerFactoryClass=com.sun.jdori.fostore.FOStorePMF
(2)javax.jdo.option.ConnectionURL=fostore:jdoriDB
(3)javax.jdo.option.ConnectionUserName=mario
(4)javax.jdo.option.ConnectionPassword=thePassword
```

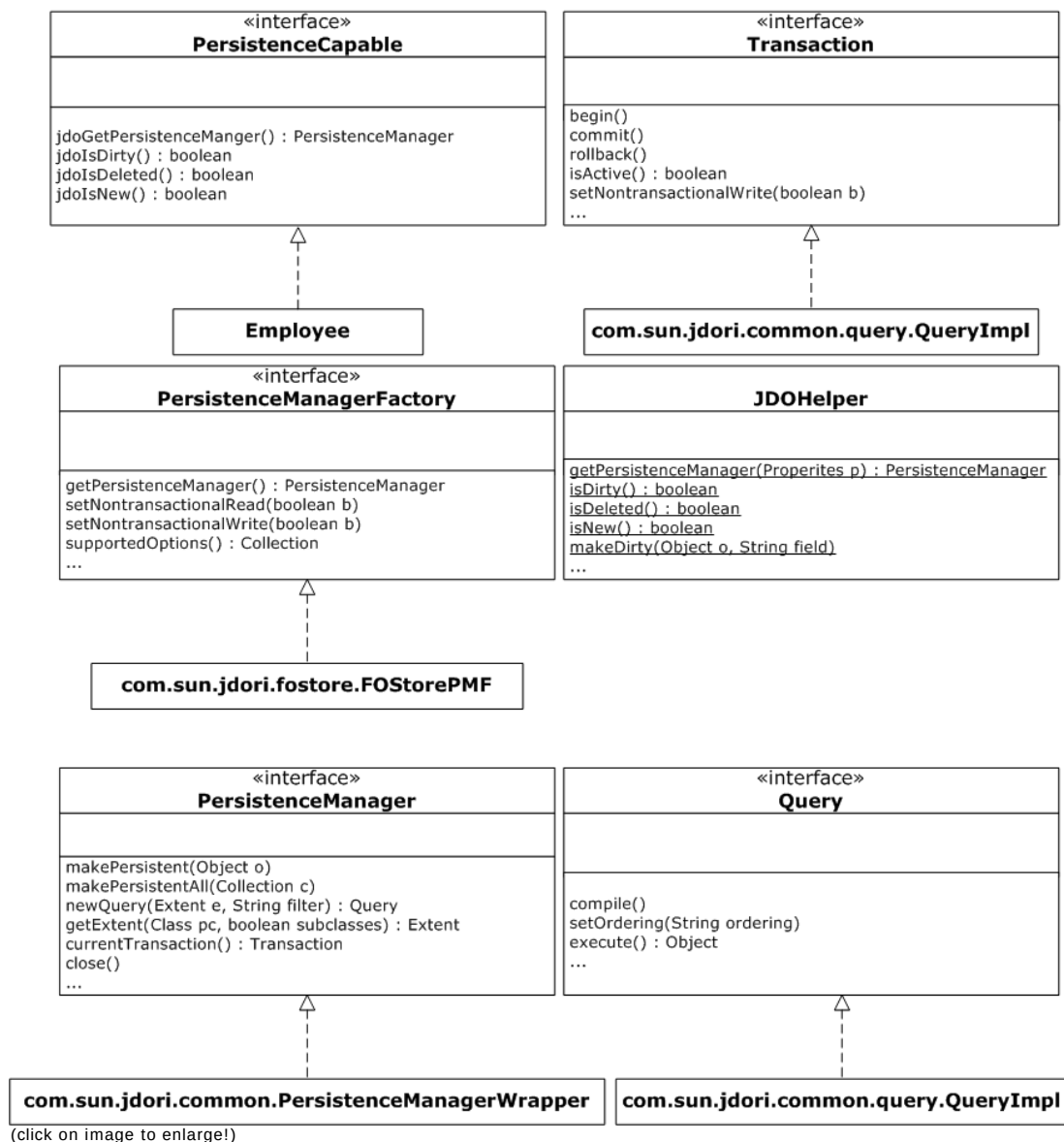
[Download des Beispiels](#)

Struktur der JDO-API

Die JDO-API ist im Rahmen des Java Community Prozesses als Java-Schnittstellensammlung nebst zugehöriger Semantikdefinition spezifiziert. Die Implementierung der Schnittstellen erfolgt durch den Anbieter der jeweiligen JDO-Implementierung und erfolgt auf den jeweiligen Persistenztyp abgestimmt.

[Abbildung 5](#) zeigt die grundlegenden Schnittstellen der JDO-API sowie die sie anbietenden Klassen der Referenzimplementierung.

Abbildung 5: Grundlegende Struktur der JDO-API



Die Schnittstelle `PersistenceCapable` bildet das Rückgrat der gesamten Persistenzbemühungen. Jede Klasse, deren Speicherung durch JDO verwaltet werden soll (in Beispiel die Klasse `Employee`) muß diese Schnittstelle zwingend implementieren.

Typischerweise erfolgt diese Implementierung jedoch nicht direkt durch den Applikationsprogrammierer, sondern wird im Rahmen der Bytecodeanreicherung nachträglich hinzugefügt.

Zur Interaktion mit Klassen, deren Implementierung der in `PersistenceCapable` deklarierten Methoden erst nach dem initialen Übersetzungsvorgang hinzugefügt werden kann der JDO-Anbieter die Hilfsklasse `JDOHelper` anbieten. Diese definiert verschiedene, ausschließlich als statisch deklarierte, Methoden um mit Objekten von Klassen zu operieren, als würden diese die Schnittstelle `PersistenceCapable` umsetzen, ohne deren Klassen zur tatsächlichen Schnittstellenimplementierung verpflichtet.

Damit stellt `JDOHelper` die unabdingbare Voraussetzung zur Anwendungsentwicklung unter Verwendung der Bytecodeanreicherung dar, da diese erst nach dem Übersetzungsvorgang Implementierungen derjenigen Schnittstellen hinzufügt, die bereits im Code verwendeten wurden. Ferner bietet die Klasse die Möglichkeit den aktuellen Persistenzzustand eines JDO-verwalteten Objektes auszulesen.

Zur Erzeugung von Objekten, die später den Zugriff auf das physische Speichermedium regeln dienen die Umsetzungen der Schnittstelle `PersistenceManagerFactory`. Sie erlaubt die Parametrisierung und Verwaltung der Verbindung zum Persistenzmedium. Bereitgestellt wird die Implementierung, im Falle der Referenzimplementierung, durch die Klasse `com.sun.jdori.fostore.FOStorePMF`.

Die Verbindung zwischen Schnittstelle und tatsächlicher Implementierung wird im Rahmen der in Beispiel 32 gezeigten JDO-Konfiguration definiert. Zum Wechsel des Persistenzanbieters -- etwa von der durch die Referenzimplementierung angebotenen dateibasierten Speicherung auf eine datenbankgestützte Umsetzung -- genügt im die Abänderung dieses Eintrages in der Konfigurationsdatei.

Klassen, welche die Schnittstelle `PersistenceManagerFactory` implementieren, werden zur Erzeugung von sog. `PersistenceManagern` herangezogen. Umsetzungen dieser Schnittstelle (im Falle der Referenzimplementierung ist dies die Klasse `com.sun.jdori.common.PersistenceManagerWrapper`) dienen zur Interaktion mit der Persistenzverwaltung innerhalb der JDO nutzenden Applikation. Alle Änderungen des Zustandes eines persistenten Objektes werden durch diese Klasse abgewickelt.

JDO wickelt sämtliche Zugriffe auf die persistenten Daten transaktionsgesichert ab. Dieser Mechanismus wird auf der abstrakten Ebene der API durch die Schnittstelle `Transaction` definiert und steht daher für alle Persistenzanbieter gleichermaßen zur Verfügung.

Die Schnittstelle definiert alle zur Transaktionssteuerung benötigten Operationen (darunter `begin`, `commit` und `rollback`) an.

Im Falle der Referenzimplementierung wird die Schnittstelle durch die Klasse `com.sun.jdori.common.query.QueryImpl` umgesetzt.

Zusätzlich sieht JDO eine abstrakte Möglichkeit zur Formulierung von Anfragen auf den verwalteten Datenbestand vor. Die notwendige Schnittstelle wird durch `Query` bereitgestellt.

Hierfür müssen die verschiedenen JDO-Implementierungen ebenfalls eigene Umsetzungen vorsehen.

Erzeugen eines persistenten Objektspeichers

Zur Erzeugung eines Objektspeichers ist bereits die Nutzung der Implementierungen der zentralen JDO-Schnittstellen sowie die der Transaktionssteuerung notwendig.

Das Beispiel zeigt die notwendigen Schritte zur Erzeugung eines persistenten Objektspeichers.

Zunächst lädt das Beispiel die Konfiguration aus der Eigenschaftsdatei des Beispiels 32.

Anschließend wird durch die mit `true` belegte implementierungsspezifische Eigenschaft `com.sun.jdori.option.ConnectionCreate` festgelegt, daß im Rahmen des Verbindungsaufbaus auch notwendigenfalls der Objektspeicher neu erzeugt wird.

Die Interaktion mit JDO beginnt durch die Erzeugung eines `PersistenceManagerFactory` konformen Objektes durch den Aufruf `getPersistenceManagerFactory` unter Auswertung der zuvor geladenen und ergänzten Konfigurationseigenschaften.

Nach der Erzeugung des Factory-Objektes kann mittels diesem durch den Aufruf `getPersistenceManager` ein Objekt erzeugt werden, das die Interaktion mit dem Objektspeicher bereitstellt. Durch die Ermittlung des Persistenzmanagers wird gleichzeitig eine Verbindung zum Persistenzanbieter aufgebaut.

Ausgehend von diesem Verwaltungsobjekt kann durch Definition einer „leeren“ Transaktion -- d.h. einer Transaktion, die jenseits der Erzeugung des transaktionalen Kontexts und seines Abschlusses mit `commit`, keine Operationen definiert -- der Objektspeicher erzeugt werden.

Den Abschluß der Interaktion mit dem Objektspeicher bildet die Beendigung der Verbindung durch Ausführung der Methode `close` des Verbindungsobjektes.

Beispiel 33: Erzeugung eines persistenten Objektspeichers

```
(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Properties;
(4)
(5)import javax.jdo.JDOHelper;
(6)import javax.jdo.PersistenceManager;
(7)import javax.jdo.PersistenceManagerFactory;
(8)import javax.jdo.Transaction;
(9)
(10)public class JDOCreatedB {
(11)    public static void main(String args[]) {
(12)        Properties props = new Properties();
(13)        try {
(14)            InputStream is = ClassLoader.getResourceAsStream("jdo.
properties");
(15)            props.load(is);
(16)            props.put("com.sun.jdori.option.ConnectionCreate","true");
(17)        } catch (IOException ioe) {
(18)            System.out.println("Error loading properties");
(19)            System.exit(1);
(20)        }
(21)        PersistenceManagerFactory pmf = JDOHelper.
getPersistenceManagerFactory(props);
(22)        PersistenceManager pm = pmf.getPersistenceManager();
(23)
(24)        Transaction tx = pm.currentTransaction();
(25)        tx.begin();
```



```

(26)         tx.commit();
(27)         pm.close();
(28)     }
(29)}

```

[Download des Beispiels](#)

Parametrisierung der Persistenz

Grundsätzlich können Ausprägungen jeder beliebigen Javaklasse durch JDO persistiert werden, solange diese Klassen die Schnittstelle `PersistentCapable` explizit im Quellcode implementieren oder die benötigte Implementierung im Rahmen der Bytecodeanreicherung hinzugefügt wird. Zur Steuerung des konkreten Persistenzverhaltens wird eine zusätzliche Konfigurationsdatei benötigt. Diese bedient sich der bekannten XML-Syntax und definiert das Persistenzverhalten der durch JDO zu verwaltenden Klasseninstanzen näher.

Beispiel 34 zeigt zunächst die zu persistierende Klasse `Employee`.

Beispiel 34: Zu persistierende Javaklasse

```

(1)package de.jeckle.jdotest;
(2)
(3)import java.util.HashSet;
(4)import java.util.Iterator;
(5)
(6)public class Employee {
(7)    private String name;
(8)    private String department;
(9)    private HashSet projects = new HashSet();
(10)
(11)    public String getName() {
(12)        return name;
(13)    }
(14)    public String getDepartment() {
(15)        return department;
(16)    }
(17)    public void setName(String name) {
(18)        this.name = name;
(19)    }
(20)    public void setDepartment(String department) {
(21)        this.department = department;
(22)    }
(23)    public void addProject(String project) {
(24)        projects.add(project);
(25)    }
(26)
(27)    public String toString() {
(28)        String result="Employee named "+name+" works in department
"+department;
(29)        result+="\nworks in: ";
(30)        Iterator i = projects.iterator();
(31)        while (i.hasNext()) {
(32)            result+=(String) i.next();
(33)            result+=", ";
(34)        }
(35)        return (result);
(36)    }
(37)}

```



[Download des Beispiels](#)

Die Nutzung JDO-gestützter Objektpersistenz impliziert keinerlei Modifikationen oder Ergänzungen am Quellcode. Ebenso sind keinerlei Umsetzungskonventionen einzuhalten, die im Beispiel definierten get- und set-Methoden dienen lediglich der vereinfachten Interaktion.

Das Beispiel 35 illustriert eine Parameterdatei zur Definition des spezifischen Persistenzverhaltens von Objekten der Klasse `Employee`.

Beispiel 35: Parametrisierung der Objektpersistenz

```

(1)<?xml version="1.0"?>
(2)<!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata
1.0//EN" "http://java.sun.com/dtd/jdo_1_0.dtd">
(3)<jdo>
(4)    <package name="de.jeckle.jdotest">
(5)        <class name="Employee">
(6)            <field
(7)                name="name"
(8)                persistence-modifier="persistent"/>
(9)            <field
(10)                name="department"
(11)                persistence-modifier="persistent"/>
(12)            <field
(13)                name="projects">
(14)                    <collection
(15)                        element-type="java.lang.String"
(16)                        embedded-element="true"/>
(17)            </field>
(18)        </class>
(19)    </package>
(20)</jdo>

```



[Download des Beispiels](#)

Die XML-Datei definiert zunächst den Paket- und Klassennamen der zu persistierenden Klasse mittels des Attributs name der XML-Elemente package und class.

Innerhalb eines class-Elements kann für jedes Attribut der Javaklasse ein mit field benanntes Element zur näheren Charakterisierung des Speicherungsverhaltens angegeben werden.

Ein solches Element trägt zunächst im Attribut name den klassenweit eindeutigen Namen des Attributs und erlaubt die Festlegung des spezifischen Persistenzverhaltes mittels der Belegung des Attributs persistence-modifier. Ist dieses mit dem Wert persistent versehen, so wird ein so gekennzeichnetes Attribut durch JDO im Datenspeicher persistiert. Trägt das XML-Attribut den Wert none, so wird das Javaattribut bei der Abbildung in den JDO-Datenspeicher ignoriert.

Zusätzlich besteht die Möglichkeit durch die Belegung mit transactional die Zwischenspeicherung des Attributwertes während der Abarbeitung einer Transaktion zu erzwingen, um so eine spätere Wiederherstellung (nach einem Aufruf von rollback) zu gewährleisten. Jedoch werden Felder, die so gekennzeichnet sind, nicht persistent in den Datenspeicher übernommen, sondern stehen nur während der Programmlaufzeit zur Verfügung.

Fehlt diese Spezifikation zu einem Attribut in der XML-Datei, so wird vorgabegemäß die Belegung mit persistent angenommen, sofern es in der beherbergenden Javaklasse nicht als static, transient oder final ausgewiesen ist.

Attribute vom Typ einer Sammlungsklasse, wie sie durch die [Collection API](#) definiert werden müssen zusätzlich mit einem collection-Element, welches innerhalb des field-Elements platziert ist, charakterisiert. Das collection-Element spezifiziert durch sein Attribut element-type den Typ der Elemente in der Sammlung festlegt. Zusätzlich kann durch das Boole'sche-Attribut embedded-element gesteuert werden, ob die Inhalte des Sammlungsobjektes zusammen mit dem die Sammlung referenzierenden Objekt persistiert werden sollen.

Das Beispiel legt für alle Attribute der Klasse Employee ihre persistente Speicherung fest (Belegung des XML-Attributs persistence-modifier für alle Attribute persistent); ebenso wird die in Objekten des Typs Employee, unter dem Namen projects, enthaltene Sammlungsinstanz einschließlich ihrer Inhaltsobjekte des Standard-API-Typs String dauerhaft abgespeichert.

Über diese Festlegungen hinaus gestattet das Parametrisierungsformat die Festlegung spezifischer Konsistenzsemantik in Gestalt der Auszeichnung eines *Primärschlüssels*. Dieses aus dem relationalen Modell bekannte Konstrukt fordert die Eindeutigkeit eines Attributs oder einer Kombination von Attributen über die gesamte Menge der Ausprägungen eines Typs.

Durch die Unterstützung als abstraktes JDO-Konstrukt steht dieses Konzept zur Konsistenzsicherung auch für Applikationen zur Verfügung, die sich nicht relationaler Datenbanken als Persistenzdienstleister bedienen.

Zur Realisierung des Primärschlüsselkonzeptes ist das als Schlüssel zu interpretierende Attribut in der XML-Beschreibung zusätzlich mit dem XML-Attribut primary-key zu versehen, welches den Wert true tragen muß. Zusätzlich ist innerhalb des Elements class diejenige Klasse anzugeben, welche das Attribut beherbergt, das als Schlüssel herangezogen werden soll.

36 zeigt die notwendigen Modifikationen an der Parameterdatei des Beispiels 35 um das Java-Attribut name als Primärschlüssel festzulegen. Die primärschlüsselanbietende Klasse ist in diesem Falle die Klasse Employee selbst, weshalb sich ihr Name auch im XML-Attribut objectid-class des

class-Elements findet.

Beispiel 36: Parametrisierung der Objektpersistenz und Definition eines Primärschlüssels

```
(1)<?xml version="1.0"?>
(2)<!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata
1.0//EN" "http://java.sun.com/dtd/jdo_1_0.dtd">
(3)<jdo>
(4)    <package name="de.jeckle.jdotest">
(5)        <class name="Employee"
(6)            objectid-class="Employee">
(7)                <field
(8)                    primary-key="true"
(9)                    name="name"
(10)                   persistence-modifier="persistent"/>
(11)            <field
(12)                name="department"
(13)                persistence-modifier="persistent"/>
(14)            <field
(15)                name="projects">
(16)                    <collection
(17)                        element-type="java.lang.String"
(18)                        embedded-element="true"/>
(19)                </field>
(20)            </class>
(21)    </package>
(22)</jdo>
```



[Download des Beispiels](#)

Konsequenz der Einführung eines Primärschlüsselattributs ist die Überwachung der damit einhergehenden Konsistenzbedingungen durch das JDO-Laufzeitsystem. So führen Versuche zwei Objekte, die sich in der Belegung des als Primärschlüssel definierten Attributs nicht unterscheiden ebenso zu Fehlern wie schreibende Zugriffe auf dergestalt ausgezeichnete Attribute.

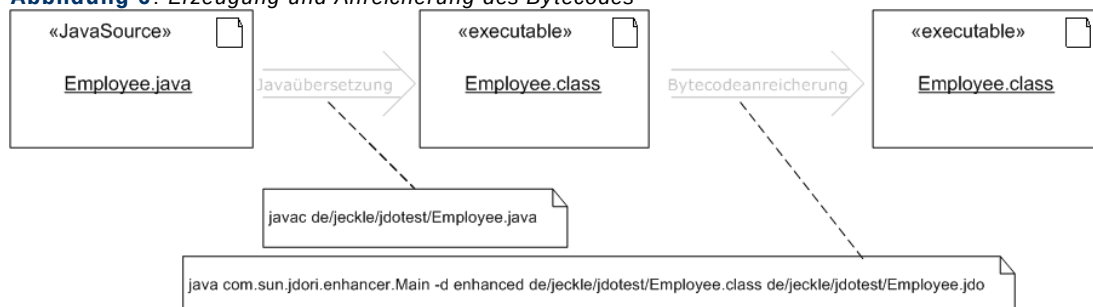
Anreicherung des Bytecodes

Voraussetzung der Persistenzverwaltung eines Objektes durch JDO ist die entsprechende Modifikation dieses Objektes, konkret die Implementierung der in PersistenceCapable festgelegten Operationen durch Methoden der objekterzeugenden Klasse.

Dies wird jedoch nur in Ausnahmefällen durch den Applikationsprogrammierer direkt vorgenommen. Häufigste Einsatzform der JDO-API ist die Anwendung der Bytecodeanreicherung, welche die Implementierung der notwendigen Funktionalität automatisiert vornimmt und diese nach dem eigentlichen Übersetzungsvorgang in den erstellten Bytecode einbringt.

[Abbildung 6](#) zeigt die daher notwendigen zwei Übersetzungsschritte.

Abbildung 6: Erzeugung und Anreicherung des Bytecodes



(click on image to enlarge!)

Die Illustration versammelt die zur Erzeugung und Anreicherung des Bytecodes der per JDO zu persistierenden Klasse Employee aus Beispiel 34. Zur Anreicherung des Bytecodes werden die in Beispiel 35 getroffenen Parametrisierungen herangezogen.

Zunächst wird der im Paket de.jeckle.jdotest abgelegte Quellcode Employee.java mit dem Javacompiler in (gewöhnlichen) Bytecode übersetzt.

Anschließend wird dieser vermöge des in der JDO-Referenzimplementierung vorhandenen Werkzeuges Enhancer um die Implementierung der in der Schnittstelle PersistenceCapable definierten Operationen angereichert. Hierzu wird dem Enhancer (bereitgestellt durch die Klasse com.sun.jdori.enhancer.Main zunächst das Zielverzeichnis des zu erzeugenden Bytecodes mittels des Parameters d übergeben. Naheliegenderweise kann der aus dem ursprünglichen Bytecode durch Erweiterung erzeugte nicht die Ausgangsdatei überschreiben, daher wird der angereicherte Bytecode im Verzeichnis enhanced gespeichert. Zusätzlich ist dem Enhancer der

vollqualifizierter Name der anzureichernden Klasse sowie der vollqualifizierte Pfad der Parameterdatei (im Beispiel: de/jeckle/jdotest/Employee.jdo) zu übergeben. Diese muß im Falle des Einsatzes der Referenzimplementierung zwingend die Extension jdo besitzen.

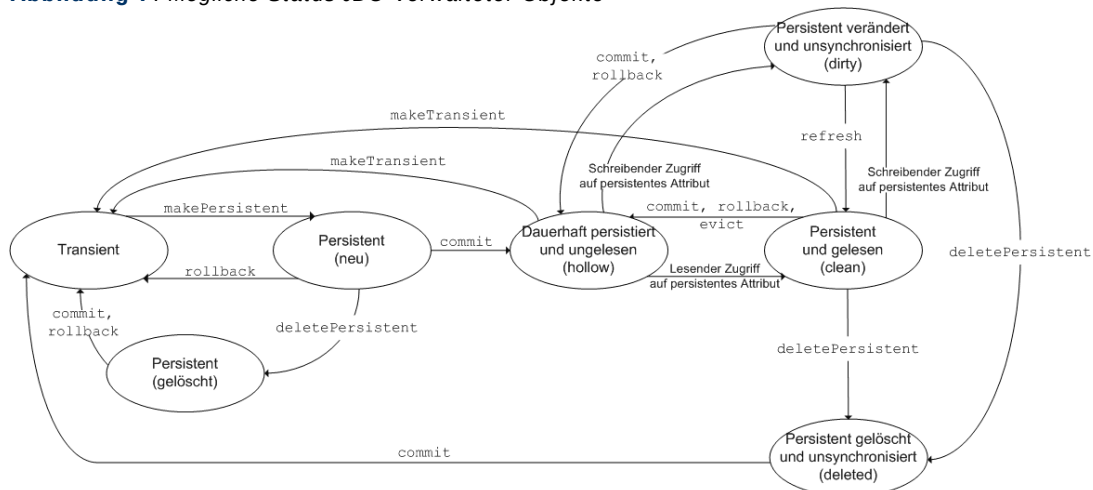
Status JDO-verwalteter Objekte

Im Zusammenspiel zwischen transienter Objektverwaltung durch die Applikation im Hauptspeicher und persistenter Objektverwaltung durch JDO im Hintergrundspeicher werden verschiedene Status eines verwalteten Objekts unterschieden zwischen denen explizite Übergänge durch API-Aufrufe vorgegeben sind bzw. implizit durch Operationen auf den involvierten Objekten bestehen. Im Detail werden folgende Status unterschieden:

- **Transient**: Instanzierte Objekte im Hauptspeicher. Hierunter fallen alle noch nicht innerhalb von Transaktionen persistierten Objekte ebenso wie unangereicherte Javaobjekte und solche die ausschließlich über Attribut verfügen, die als in der XML-Parametrisierungsdatei als transient gekennzeichnet sind.
- **Persistent (neu)**: Objekte, die innerhalb einer laufenden (d.h. weder durch commit noch rollback abgeschlossenen) Transaktion erzeugt wurden.
Endet eine Transaktion, etwa durch Programmabbruch, in diesem Zustand, so werden die Objekte mit diesem Status nicht dauerhaft gespeichert.
- **Persistent (gelöscht)**: Objekte, die in einer noch nicht abgeschlossenen Transaktion persistiert und anschließend gelöscht wurden.
Endet eine Transaktion, etwa durch Programmabbruch, in diesem Zustand, so werden die Objekte mit diesem Status nicht dauerhaft gespeichert, da sowohl der Persistierungs- als auch der anschließende Löschvorgang noch nicht durch commit bestätigt wurden.
- **Dauerhaft persistiert und ungelesen (hollow)**: Objekte, die durch Abschluß einer Transaktion mit commit dauerhaft gespeichert wurden und auf die noch kein Zugriff (weder lesend noch schreibend) erfolgte.
- **Persistent und gelesen (clean)**: Objekte, die persistent im Hintergrundspeicher abgelegt wurden und auf die bisher lediglich lesende Zugriffe erfolgten.
- **Persistent verändert und unsynchronisiert (dirty)**: Persistentes Objekt, dessen Inhalt im Rahmen einer noch nicht abgeschlossenen Transaktion verändert wurde.
Wurden zwar Attributinhalt eines persistenten Objektes verändert, jedoch keine Wertänderungen vorgenommen, d.h. in ein Attribut wird mit demselben Wert belegt, den es bereits enthält, dann steht es dem JDO-Implementierer frei diese Schreiboperation so zu implementieren, daß nicht der Zustand *dirty* eingenommen wird.
Zusätzlich kann jedes Objekt durch Aufruf der API-Methode `makeDirty` manuell in diesen Zustand versetzt werden.
Als Folge der Schreiboperation im Hauptspeicher differieren dessen Inhalte von denen des persistenten Objektspeichers.
Durch Aufruf der API-Methode `refresh` werden die Inhalte von Haupt- und Hintergrundspeicher synchronisiert, d.h. Inhalte des Hintergrundspeichers werden in den Hauptspeicher übernommen.
- **Persistent gelöscht und unsynchronisiert (deleted)**: Persistentes Objekt, das im Rahmen einer noch nicht abgeschlossenen Transaktion gelöscht wurde.
Als Folge der Löschoperation im Hauptspeicher differieren dessen Inhalte von denen des persistenten Objektspeichers und alle Leseoperationen auf nicht-Primärschlüsselfelder führen zu Laufzeitfehlern.

[Abbildung 7](#) zeigt die verschiedenen JDO-Status sowie die Ereignisse, die zu Zustandsübergängen führen, in der Übersicht.

Abbildung 7: Mögliche Status JDO-verwalteter Objekte



Speicherung von Objekten

Zur Speicherung von Objekten, deren Klassen durch den Bytecodeanreicherungsprozeß nachbearbeitet wurden, bietet die JDO-API die Aufrufe `makePersistent` und `makePersistentAll` an. Diese werden innerhalb eines Transaktionskontextes als Methoden eines `PersistenceManager`-Objekts ausgeführt.

Beispiel 37 zeigt die Speicherung von drei Objekten der Klasse `Employee`, deren übersetzter Bytecode durch Anreicherung zur JDO-Kompatibilität modifiziert wurde.

Zunächst wird mit `empCol` vom Standardtyp `Vector` eine Sammlungsobjekt zur Aufnahme von Objektreferenzen definiert. Dieser Objektsammlung werden die Referenzen auf die erzeugten `Employee`-Objekte (`emp1`, `emp2` und `emp3`) hinzugefügt.

Als Voraussetzung der Interaktion mit dem Objektspeicher muß zunächst eine Transaktion eröffnet werden. Hierzu muß zunächst durch Aufruf der Methode `currentTransaction` die der `PersistenceManager`-Instanz zugeordnete Transaktion ermittelt werden. Ausgehend vom gelieferten Ergebnisobjekt kann durch Ausführung der Methode `begin` eine neuer Transaktionskontext eröffnet werden.

Der Aufruf von `makePersistentAll` persistiert bei Übergabe der Objektsammlung alle in der Sammlung referenzierten Objekte. Alternativ können Einzelobjekte durch die Methode `makePersistent` in den Zustand dauerhafter Speicherung überführt werden.

Zur Übernahme in den Hintergrundspeicher muß der Transaktionskontext durch Aufruf von `commit` abgeschlossen werden. Der Aufruf von `rollback` würde stattdessen alle in der Transaktion vorgenommenen Änderungen verwerfen und auf den im Hintergrundspeicher verwalteten Datenzustand zurückgesetzt.

Beispiel 37: Speicherung von Objekten mit JDO

```
(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Properties;
(4)import java.util.Vector;
(5)
(6)import javax.jdo.JDOHelper;
(7)import javax.jdo.PersistenceManager;
(8)import javax.jdo.PersistenceManagerFactory;
(9)
(10)import de.jeckle.jdotest.Employee;
(11)
(12)public class JDOStoreObj {
(13)    public static void main(String args[]) {
(14)        Properties props = new Properties();
(15)        try {
(16)            InputStream is =
(17)                ClassLoader.getResourceAsStream("jdo.
properties");
(18)            props.load(is);
(19)        } catch (IOException ioe) {
(20)            System.out.println("Error loading properties");
(21)            System.exit(1);
(22)        }
(23)        PersistenceManagerFactory pmf =
(24)            JDOHelper.getPersistenceManagerFactory(props);
(25)        PersistenceManager pm = pmf.getPersistenceManager();
(26)
(27)        Vector empCol = new Vector();
(28)
(29)        Employee emp1 = new Employee();
(30)        emp1.setName("Mario Jeckle");
(31)        emp1.setDepartment("D001");
(32)        emp1.addProject("P001");
(33)        emp1.addProject("P002");
(34)        empCol.add(emp1);
(35)
(36)        Employee emp2 = new Employee();
(37)        emp2.setName("John DoeX");
(38)        emp2.setDepartment("D003");
(39)        emp2.addProject("P001");
(40)        emp2.addProject("P042");
(41)        empCol.add(emp2);
(42)
(43)        Employee emp3 = new Employee();
(44)        emp3.setName("John Doe");
(45)        emp3.setDepartment("B042");
```




```

(46)         empCol.add(emp3);
(47)
(48)         Employee emp4 = new Employee();
(49)         emp4.setName("Barnie Bar");
(50)         emp4.setDepartment("B042");
(51)
(52)         pm.currentTransaction().begin();
(53)         pm.makePersistentAll(empCol);
(54)         pm.makePersistent(emp4);
(55)
(56)         pm.currentTransaction().commit();
(57)         pm.close();
(58)     }
(59)}

```

[Download des Beispiels](#)

Würde wie im Beispiel 36 gezeigt das Attribut name der Klasse Employee als Primärschlüssel definiert sein, so würde der Persistierungsversuch des durch emp3 referenzierten Objektes einen Laufzeitfehler liefern, da mit emp2 bereits ein Objekt mit derselben Belegung des Attributs name persistiert wurde.

Rücksetzen von Transaktionen

Treten während der Interaktion mit dem Persistenzspeicher, d.h. während eines noch nicht mit commit abgeschlossenen Transaktionskontextes Fehler auf, so können durch Aufruf der Methode rollback alle im aktuellen Kontext vorgenommen Änderungen auf den Stand vor Beginn der Transaktion zurückgesetzt werden.

Beispiel 38 zeigt das Verhalten der Methode rollback am Beispiel. Durch die Schreiboperation innerhalb der geöffneten Transaktion wird der Wert des Attributs name zwar verändert, jedoch durch Aufruf von rollback wieder auf den ursprünglichen Wert zurückgesetzt.

Beispiel 38: Transaktionen mit JDO

```

(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Iterator;
(4)import java.util.Properties;
(5)
(6)import javax.jdo.JDOHelper;
(7)import javax.jdo.PersistenceManager;
(8)import javax.jdo.PersistenceManagerFactory;
(9)
(10)import de.jeckle.jdotest.Employee;
(11)
(12)public class JDORollback {
(13)
(14)    public static void main(String args[]) {
(15)        Properties props = new Properties();
(16)        try {
(17)            InputStream is =
(18)                ClassLoader.getResourceAsStream("jdo.
properties");
(19)            props.load(is);
(20)        } catch (IOException ioe) {
(21)            System.out.println("Error loading properties");
(22)            System.exit(1);
(23)        }
(24)        PersistenceManagerFactory pmf =
(25)            JDOHelper.getPersistenceManagerFactory(props);
(26)        PersistenceManager pm = pmf.getPersistenceManager();
(27)
(28)        Employee e = new Employee();
(29)        e.setName("Marta Mayer");
(30)
(31)        pm.currentTransaction().begin();
(32)        pm.makePersistent(e);
(33)        pm.currentTransaction().commit();
(34)        displayPersistedObjects(pm);

```



```

(35)
(36)         System.out.println("Martha gets married and changes her name");
(37)
(38)         pm.currentTransaction().begin();
(39)         e.setName("Marta Smith");
(40)         pm.makePersistent(e);
(41)         displayPersistedObjects(pm);
(42)         System.out.println("Suppose and error happens now ...\\nRolling
back");
(43)         pm.currentTransaction().rollback();
(44)
(45)         displayPersistedObjects(pm);
(46)     }
(47)     private static void displayPersistedObjects(PersistenceManager pm) {
(48)         Iterator i = pm.getExtent(Employee.class, false).iterator();
(49)         while (i.hasNext()) {
(50)             System.out.println((Employee) i.next());
(51)         }
(52)     }
(53)}

```

[Download des Beispiels](#)

Die Ausführung des Beispiels liefert folgende Ausgabe:

```

Employee named Marta Mayer works in department null
works in projects:
Martha gets married and changes her name
Employee named Marta Smith works in department null
works in projects:
Suppose and error happens now ...
Rolling back
Employee named Marta Mayer works in department null
works in projects:

```

Schreiboperationen ohne Transaktionsschutz

Ist in bestimmten Anwendungsfällen die Arbeit ohne Transaktionsschutz -- und damit ohne die Möglichkeit der expliziten Rücksetzung von Änderungen mittels rollback oder der impliziten Rücksetzung nach einem Systemausfall -- gewünscht, so kann dies durch Aktivierung der Schreibfunktionalität ohne Transaktionsschutz erreicht werden.

Hierzu muß die Methode `setNontransactionalWrite` mit dem Übergabeparameter `true` für eine Transaktion aufgerufen werden.

Das nachfolgende Beispiel zeigt als Modifikation von Beispiel 37 die persistente Übernahme einer Wertänderung ohne Transaktionsschutz.

Beispiel 39: Schreiboperation ohne Transaktionsschutz

```

(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Iterator;
(4)import java.util.Properties;
(5)
(6)import javax.jdo.JDOHelper;
(7)import javax.jdo.PersistenceManager;
(8)import javax.jdo.PersistenceManagerFactory;
(9)
(10)import de.jeckle.jdotest.Employee;
(11)public class JDONonTransact {
(12)    public static void main(String args[]) {
(13)        Properties props = new Properties();
(14)        try {
(15)            InputStream is =
(16)                ClassLoader.getResourceAsStream("jdo.
properties");
(17)            props.load(is);
(18)        } catch (IOException ioe) {
(19)            System.out.println("Error loading properties");
(20)            System.exit(1);

```




```

(21)    }
(22)    PersistenceManagerFactory pmf =
(23)        JDOHelper.getPersistenceManagerFactory(props);
(24)    PersistenceManager pm = pmf.getPersistenceManager();
(25)
(26)    Employee e = new Employee();
(27)    e.setName("Marta Mayer");
(28)    pm.currentTransaction().setNontransactionalWrite(true);
(29)
(30)    pm.currentTransaction().begin();
(31)    pm.makePersistent(e);
(32)    pm.currentTransaction().commit();
(33)    displayPersistedObjects(pm);
(34)
(35)    //martha gets married and changes her name
(36)
(37)    e.setName("Marta Smith");
(38)
(39)    displayPersistedObjects(pm);
(40)    }
(41)    private static void displayPersistedObjects(PersistenceManager pm) {
(42)        Iterator i = pm.getExtent(Employee.class, false).iterator();
(43)        while (i.hasNext()) {
(44)            System.out.println((Employee) i.next());
(45)        }
(46)    }
(47)}

```

[Download des Beispiels](#)

Traversierung des persistenten Objektbestandes

Zugriffe auf alle im Hintergrundspeicher verwalteten Objekte werden ebenfalls einheitlich durch Methoden der Implementierung der Schnittstelle `PersistenceManager` abgewickelt. Zur Traversierung des vollständigen Bestandes aller Instanzen einer Klasse bietet diese Schnittstelle die Operation `getExtent` an. Sie liefert alle Elemente der Extension (d.h. der Gesamtheit von Ausprägungen) einer gegebenen Klasse.

Beispiel 40 zeigt die Verwendung der Methode. Als Parameter wird diejenige Klasse übergeben, deren Ausprägungen zu ermitteln sind. Zusätzlich kann durch einen Boole'schen Schalter gesteuert werden, ob auch Subklassen der übergebenen Klasse retourniert werden sollen.

Der Aufruf liefert eine Sammlung von Objekten des Typs, welcher der Methode `getExtent` übergeben wurde.

Beispiel 40: Traversierung des Objektbestandes

```

(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Iterator;
(4)import java.util.Properties;
(5)
(6)import javax.jdo.JDOHelper;
(7)import javax.jdo.PersistenceManager;
(8)import javax.jdo.PersistenceManagerFactory;
(9)
(10)import de.jeckle.jdotest.Employee;
(11)
(12)public class JDOListObj {
(13)    public static void main(String args[]) {
(14)        Properties props = new Properties();
(15)        try {
(16)            InputStream is =
(17)                ClassLoader.getResourceAsStream("jdo.
properties");
(18)            props.load(is);
(19)        } catch (IOException ioe) {
(20)            System.out.println("Error loading properties");
(21)            System.exit(1);
(22)        }
(23)        PersistenceManagerFactory pmf =
(24)            JDOHelper.getPersistenceManagerFactory(props);

```



```

(25)         PersistenceManager pm = pmf.getPersistenceManager();
(26)
(27)         Iterator i = pm.getExtent(Employee.class, false).iterator();
(28)         while (i.hasNext()) {
(29)             System.out.println((Employee)i.next());
(30)         }
(31)     }
(32)}

```

[Download des Beispiels](#)

Anfragen an den persistenten Objektbestand

Als mächtige Alternative zur manuellen Traversierung einer Objektextension spezifiziert JDO die Verwendung einer eigenen Anfragesprache auf Basis des Standards der *Object Query Language* (OQL) der Object Database Management Group (ODMG).

Diese -- als *JDO Object Query Language* (JDOQL) bezeichnete -- Anfragesprache ist direkt in die JDO-API integriert und wird über verschiedene Einzelmethoden genutzt. Aus diesem Grunde sind JDOQL-Anfragen nicht direkt mit den konsizsen SQL- oder OQL-Anfragen vergleichbar.

Beispiel 41 zeigt die Einbettung der Anfragesprache in die JDO-API.

Beispiel 41: Anfrage auf den persistenten Objektbestand mittels OQL

```

(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Collection;
(4)import java.util.Iterator;
(5)import java.util.Properties;
(6)
(7)import javax.jdo.Extent;
(8)import javax.jdo.JDOHelper;
(9)import javax.jdo.PersistenceManager;
(10)import javax.jdo.PersistenceManagerFactory;
(11)import javax.jdo.Query;
(12)
(13)import de.jeckle.jdotest.Employee;
(14)public class JDOQuery {
(15)    public static void main(String args[]) {
(16)        Properties props = new Properties();
(17)        try {
(18)            InputStream is =
(19)                ClassLoader.getResourceAsStream("jdo.
properties");
(20)            props.load(is);
(21)        } catch (IOException ioe) {
(22)            System.out.println("Error loading properties");
(23)            System.exit(1);
(24)        }
(25)        PersistenceManagerFactory pmf =
(26)            JDOHelper.getPersistenceManagerFactory(props);
(27)        PersistenceManager pm = pmf.getPersistenceManager();
(28)
(29)        Extent ext = pm.getExtent(Employee.class, false);
(30)        String filter = "department == \"B042\"";
(31)        Query qry = pm.newQuery(ext, filter);
(32)        qry.setOrdering("name ascending");
(33)        qry.compile();
(34)        Collection c = (Collection) qry.execute();
(35)
(36)        Iterator i = c.iterator();
(37)        while (i.hasNext()) {
(38)            System.out.println(i.next());
(39)        }
(40)    }
(41)
(42)}

```



[Download des Beispiels](#)

Das Beispiel illustriert eine Anfrage, die alle Employee-Objekte liefert, deren department-Attribut mit dem Wert B042 belegt ist und liefert die nach dem Inhalt des Attributes name in aufsteigender Reihenfolge sortiert.

Hierzu wird zunächst die vollständige Extension der Klasse Employee ermittelt. Allerdings Extrahiert dieser Aufruf noch keine Werte aus dem persistenten Objektspeicher, sondern schafft nur die Grundlagen einer späteren manuellen Traversierung oder der Anfrage via JDOQL.

Zur Vorbereitung der tatsächlichen physischen Anfrage wird zunächst eine Zeichenkette geeignet belegt, um als Filterausdruck dienen zu können, der auf die vollständige Extension angewandt wird. Im Beispiel ist dieser Filterausdruck mit `department == \"B042\"` belegt. Aus Gründen der Zeichenkettenverarbeitung in Java muß hierzu der notwendige Einschluß des zu suchenden Wertes in Anführungszeichen geeignet maskiert werden.

Nach diesen Vorbereitungsschritten kann durch den Aufruf der durch das PersistenceManager-kompatible Objekt bereitgestellten Methode `newQuery` ein neues Anfrageobjekt (vom Typ `Query`) erzeugt werden.

Dieses Objekt erlaubt nach der gezeigten Festlegung des Anfrageumfanges die Parametrisierung der Anfrage. Das Beispiel illustriert dies am Aufruf der Methode `setOrdering`, die es erlaubt eine bestimmte Sortierreihenfolge der gelieferten Ergebnisse vorzugeben.

Zusätzlich kann durch die optionale Ausführung der Methode `compile` eine Prüfung der zusammengestellten Anfrage erfolgen, die zusätzlich auch interne implementierungsspezifische Optimierungen vornehmen kann.

Abschließend erfolgt die Ausführung der Anfrage durch Aufruf der Methode `execute`, welche die Anfrageergebnisse konform zur Standardschnittstelle `Collection` zurückliefert.

Löschen von Objekten

Zur Entfernung eines Objektes aus dem Objektspeicher stellt die Schnittstelle `PersistenceManager` die Methode `deletePersistent` zur Verfügung, welche ein einzelnes hauptspeicherresidentes Objekt aus dem persistenten Speicher löscht, bzw. mit `deletePersistentAll` eine Möglichkeit alle durch eine Sammlung referenzierten Objekte zu entfernen.

Da es sich hierbei um einen schreibenden Zugriff handelt, muß dieser in einen Transaktionskontext eingebettet werden oder explizit transaktionslos durchgeführt werden wie in Beispiel 39 gezeigt.

Beispiel 42 zeigt die Löschung unter Verwendung eines Transaktionskontextes.

Beispiel 42: Löschen eines Objektes aus dem persistenten Objektbestand

```
(1)import java.io.IOException;
(2)import java.io.InputStream;
(3)import java.util.Collection;
(4)import java.util.Properties;
(5)
(6)import javax.jdo.Extent;
(7)import javax.jdo.JDOHelper;
(8)import javax.jdo.PersistenceManager;
(9)import javax.jdo.PersistenceManagerFactory;
(10)import javax.jdo.Query;
(11)
(12)import de.jeckle.jdotest.Employee;
(13)
(14)public class JDODeleteObj {
(15)    public static void main(String args[]) {
(16)        Properties props = new Properties();
(17)        try {
(18)            InputStream is =
(19)                ClassLoader.getResourceAsStream("jdo.
(20)properties");
(21)            props.load(is);
(22)        } catch (IOException ioe) {
(23)            System.out.println("Error loading properties");
(24)            System.exit(1);
(25)        }
(26)        PersistenceManagerFactory pmf =
(27)            JDOHelper.getPersistenceManagerFactory(props);
(28)        PersistenceManager pm = pmf.getPersistenceManager();
(29)
(30)        Extent ext = pm.getExtent(Employee.class, false);
(31)        String filter = "name == \"Marta Smith\"";
(32)        Query qry = pm.newQuery(ext, filter);
(33)        Collection c = (Collection) qry.execute();
```



```

(34)          pm.currentTransaction().begin();
(35)          pm.deletePersistentAll(c);
(36)          pm.currentTransaction().commit();
(37)          System.out.println("Object deleted");
(38)      }
(39)}

```

[Download des Beispiels](#)

Migration zu einem anderen Persistenzdienstleister

Der JDO-Ansatz tritt mit dem Versprechen auf vollständig sowohl unabhängig vom verwendeten Persistenzmedium (etwa: Datenbank, Dateisystem, etc.) als auch der eingesetzten JDO-Implementierung zu sein. Diese Zielsetzung wird nachfolgend auf Basis des im vorhergehenden diskutierten Employee-Beispiels untersucht. Hierzu wird die frei verfügbare JDO-Implementierung *TJDO* eingesetzt, welche verschiedene Datenbankmanagementsysteme zur Speicherung der Javaobjekte heranziehen kann. Im Beispiel wird das DBMS *MySQL* Persistierung der Applikationsobjekte genutzt.

Zur Portierung der bestehenden Applikation ist lediglich die Anpassung der JDO-Eigenschaften (Property-Datei) vorzunehmen, um den neuen Persistenzdienstleister sowie die verschiedenen DBMS-Spezifika zu berücksichtigen.

Beispiel 43 zeigt die neuen Inhalte.

Beispiel 43: Konfiguration der JDO-Implementierung TJDO



```

(1)javax.jdo.PersistenceManagerFactoryClass=com.triactive.jdo.
PersistenceManagerFactoryImpl
(2)javax.jdo.option.ConnectionURL=jdbc:mysql://localhost/jdotest/
(3)javax.jdo.option.ConnectionDriverName=com.mysql.jdbc.Driver
(4)javax.jdo.option.ConnectionUserName=mario
(5)javax.jdo.option.ConnectionPassword=thePassword
(6)com.triactive.jdo.autoCreateTables=true

```

[Download des Beispiels](#)

Zunächst werden die bereits in der Konfiguration der Referenzimplementierung durch Beispiel 32 genutzten Eigenschaften zur Identifikation derjenigen Klasse, welche die JDO-Schnittstelle *PersistenceManagerFactory* implementiert sowie zur Festlegung der Verbindungs-URL und des zu verwendenden Benutzernamens und Passwortes an die neuen Gegebenheiten adaptiert. Konkret wird die durch TJDO bereitgestellte Klasse *com.triactive.jdo.PersistenceManagerFactoryImpl* als *PersistenceManagerFactory* konforme Implementierung sowie die Identifikation der zu verwendenden Datenbank nebst Benutzername und Anmeldekennwort bekanntgegeben. Zusätzlich wird mit *com.triactive.jdo.autoCreateTables* eine implementierungsspezifische Eigenschaft mit *true* belegt, die TJDO veranlaßt im Bedarfsfalle benötigte Tabellenstrukturen automatisiert zu erzeugen.

Zusätzlich erfordert die verwendete JDO-Implementierung die Adaption der im Rahmen des Bytecodeanreicherungsprozesses herangezogenen Konfigurationsdatei (Beispiel 44). Auf diesem Wege wird dem Programmierer die Möglichkeit eröffnet die Abbildung auf relationale Tabellenstrukturen beeinflussen. In der Konsequenz erfordert der Wechsel der JDO-Implementierung die Wiederholung des Anreicherungslaufes für den Bytecode der zu persistierenden Klassen.

Beispiel 44: Parametrisierung der Objektpersistenz



```

(1)<?xml version="1.0"?>
(2)  <!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata
1.0//EN" "http://java.sun.com/dtd/jdo_1_0.dtd">
(3)  <jdo>
(4)    <package name="de.jeckle.jdotest">
(5)      <class name="Employee">
(6)        <field name="name">
(7)          <extension vendor-name="triactive" key="length" value="max 32"/>
(8)        </field>
(9)        <field name="department">
(10)         <extension vendor-name="triactive" key="length" value="max 32"/>
(11)      </field>

```

```

(12)         </class>
(13)     </package>
(14) </jdo>

```

[Download des Beispiels](#)

Weitere Änderungen an den zu persistierenden Klassen oder den mit deren Objekten operierenden Applikationen ist nicht notwendig, alle Zugriffe werden nach den oben beschriebenen Änderungen transparent und ohne Neuübersetzung datenbankbasiert abgewickelt.

Vergleich der verschiedenen Persistenzansätze

Abschließend seien die charakteristischen Eigenschaften der drei diskutierten Persistenzansätze JDBC, EJB und JDO kurz vergleichend nebeneinandergestellt.

Merkmal	JDBC	EJB	JDO
Transaktionsunterstützung	✓	✓	✓
Anfragemöglichkeit	✓	✓	✓
Standardisiertes API	✓	✓	✓
Standardanfragesprache	SQL	SQL/EJBQL	JDOQL
Unterstützte Hintergrundspeicher	RDBMS	RDMBS Integrationsmiddleware	RDBMS, ORDBMS Integrationsmiddleware, Dateisystem, bel. andere
Transparenter Zugriff auf persistierte Daten		✓	✓
Berücksichtigung existierender relationaler Strukturen	✓	bei bean managed persistence	allerdings nicht im Standard vorgesehen

Die Tabelle zeigt klar, daß alle drei Persistenzmechanismen grundlegende Eigenschaften teilen, sich jedoch auch in zentralen Charakteristika unterscheiden.

Während sowohl JDBC als auch EJBs die direkte Verwendung von SQL-Anfragen gestatten bietet JDO mit JDOQL eine eigenständige Anfragesprache, die direkt in die Sprach-API eingebettet ist. Für EJBs existiert neben den in Kapitel 1.2 gezeigten Mechanismen auch die Möglichkeit der Verwendung der EJB-spezifischen Anfragesprache *EJBQL*, die jedoch hier nicht betrachtet wurde. Hinsichtlich der jeweils unterstützten Hintergrundspeicherarchitekturen zur Realisierung der Persistenz treten jedoch deutliche Unterschiede zu Tage. So ist der Einsatz der JDBC-API auf relationale Datenquellen, bzw. Datenquellen die eine relationale Sicht anbieten, beschränkt. Innerhalb der EJB-Architektur können hingegen neben den -- hier diskutierten JDBC-basierten Mechanismen -- auch die Dienste einer Integrationsmiddleware zu Speicherung herangezogen werden und so eine gewisse Unabhängigkeit vom physischen Speichermedium erreicht werden. Einzig JDO bietet durch seine starke Abstraktion die Möglichkeit beliebige Persistenzdienstleister zu nutzen.

Zur effizienten Abwicklung dieses speicherformunabhängigen Zugriffs etabliert JDO notwendigerweise eine stark abstrahierte API, deren Funktionen keinerlei Rückschlüsse auf den verwendeten Persistenzmechanismus zulassen. Für EJB läßt sich dies prinzipiell auch realisieren, allerdings müssen für die Variante der bean managed persistence innerhalb der Entity Bean die Interaktionen mit dem Persistenzdienstleister expliziert werden, beispielsweise durch JDBC. Daher verhält sich dieser Ansatz intern ähnlich zur direkten Verwendung der JDBC-API, die inhärent jeden angebotenen Persistenzmechanismus mit relationaler Zugriffssemantik belegt.

Aufgrund des vorherrschenden relationalen Speicherparadigmas kann die Einbindung bestehender Tabellenstrukturen in den API-Mechanismus gewünscht sein. Dies ist ausschließlich mit Ansätzen möglich, welche die anwenderdefinierte Strukturierung der Zugriffsausdrücke -- etwa durch die Verwendung von SQL -- gestatten. Dies ist ausschließlich für JDBC und EJB (sofern bean managed persistence verwendet wird) möglich; JDO sieht dies generell nicht vor.

Abbildung 8: Vergleich zwischen den diskutierten Persistenztechniken

Abschließend lassen sich die vorgestellten Schnittstellen hinsichtlich ihrer Möglichkeiten zur Bereitstellung eines transparenten Zugriffs auf den Hintergrundspeicher und der manuellen Eingriffsmöglichkeiten zur Kontrolle der Persistenz durch den Programmierer kategorisieren. Prinzipiell läßt sich festhalten, daß diese Eigenschaftstypen konkurrierende Zielsetzungen darstellen. So bietet JDBC zweifelsohne die größten Möglichkeiten zum steuernden Eingriff durch den Programmierer, wobei dieser Ansatz in der Interaktion auch die größte Menge Wissen des Programmierers über die etablierten Speicherstrukturen erfordert. Daher realisiert JDBC generell die geringste Transparenz im Zugriff auf den Objektspeicher. Auf der anderen Seite realisiert JDO die größtmögliche Transparenz im Objektzugriff, wobei dieser Freiheitsgrad zu generell zu Lasten der Eingriffsmöglichkeiten durch den Programmierer umgesetzt werden.

Web-Referenzen 2: Weiterführende Links



- [TJDO -- eine freie JDO-Implementierung](#)
- [JDO @ SUN](#)
- [JDOCentral.com -- Die Anlaufstelle der JDO-Entwickler](#)
- [JDO-Spezifikation](#)

▲ Architekturmuster and Umsetzungstechniken

Neben den Basistechniken und bisher vorgestellten Schnittstellen zur Realisierung von Persistenz finden gegenwärtig eine Reihe von Architekturmustern und Handreichungen zur Umsetzung der Verbindung von dauerhafter Datenspeicherung und Anwendungsprogrammierung Einsatz. Ziel dieses Kapitels ist es, ausgewählte Muster an Beispielen vorzustellen und ihren Einsatz in den Kontext der im [Abschnitt eins](#) eingeführten Schnittstellentechniken zu stellen. Hierzu werden die Muster zunächst in drei Abschnitte gegliedert und gleichzeitig im Hinblick auf die durch sie angesprochene Problemdomäne kategorisiert.

- Domänenlogik: Muster dieser Klasse dienen dazu, die Verbindung zwischen Applikationslogik und persistenter Datenspeicherung zu strukturieren und die beiden Systemebenen wirkungsvoll zu entkoppeln.
- Datenzugriff: Muster dieser Klasse dienen dazu, den Zugriff auf die verwalteten Daten zu abstrahieren und diese in einheitlicher Weise zugreifbar werden zu lassen.
- Objekt-Relational-Abbildung und -Interoperabilität: Muster dieser Klasse dienen dazu, eine Brücke zwischen objektorientierter Applikationsdatenhaltung und heute (noch) vorherrschender relationaler Speicherung zu schlagen.

Die vorgestellten Muster und Beispiele orientieren sich an den im [Buch Patterns of Enterprise Application Architecture](#) von M. Fowler vorgestellten.

Domänenlogik

Die nachfolgend eingeführten Muster zur Abbildung von Datenbank-gestützt operierender Applikationslogik in Programmstrukturen. Durch die Anwendung der diskutierten Mechanismen wird

eine Entkopplung zwischen Datenbankoperationen und Domänen-induzierten Operationen der Applikationsebene angestrebt, um diese voneinander separiert entwickeln und modifizieren zu können.

Transaction Script

Motivation und Grundidee: Zur Abbildung komplexer Geschäftsoperationen sind in der Regel eine Reihe von eigenständigen Datenbankinteraktionen notwendig. Werden einzelne dieser Interaktionen außerhalb ihres logischen Kontexts ausgeführt, so kann dies zu Inkonsistenzen des verwalteten Datenbestandes führen.

Gleichzeitig setzt die korrekte Ausführung der einzelnen Datenbankoperationen die Kenntnis der Abbildung des Geschäftsprozesses auf die technischen Strukturen voraus um gültige Abläufe konstruieren zu können.

Ziel der Anwendung der *Transaktionsskripte* (engl. *transaction script*) ist es jeden Geschäftsablauf durch genau eine Applikationsmethode abzubilden, welche die notwendigen Datenbankinteraktionen zuverlässig kapselt.

Hierbei impliziert der Terminus der *Transaktion* nicht zwingend die Nutzung von Datenbanktransaktionen im Sinne der ACID-Prinzipien.

Struktur: Typischerweise werden die Geschäftsoperationen durch eine oder mehrere Domänenklassen, d.h. Klassen deren Struktur und Logik nicht auf Basis technischer Erwägungen und Notwendigkeiten gebildet wurde, zur Verfügung gestellt. Diese Klassen treten als Dienstleister gegenüber dem Applikationsprogramm auf.

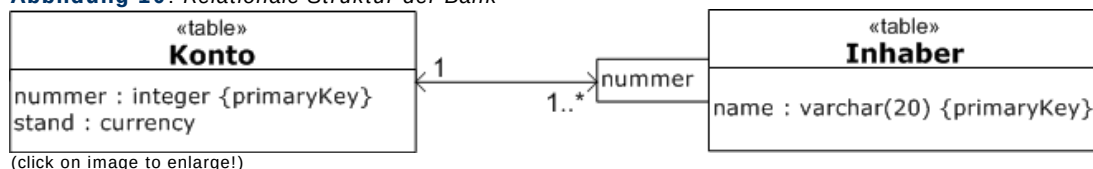
[Abbildung 9](#) zeigt die Grundstruktur des Dienstangebotes einer Bank. Sie bietet Überweisungen eines Geldbetrages von einem Konto zum anderen sowie die Möglichkeit der Erstellung einer Vermögensübersicht für einen Kunden an.

Abbildung 9: Dienstangebot der Bank



Die verwalteten Daten seien in diesem Beispiel in einer relationalen Datenbank abgelegt, die aus zwei Tabellen (Konto und Inhaber) besteht. [Abbildung 10](#) zeigt den Datenbankaufbau.

Abbildung 10: Relationale Struktur der Bank



Der Aufbau der Datenbank und ihre Befüllung mit Beispieldaten geschieht durch das folgende SQL-Script:

Beispiel 45: Erzeugung und Befüllung der Datenbank mit SQL

```

(1)CREATE TABLE konto(nummer INTEGER PRIMARY KEY, stand DECIMAL(5,2) NOT NULL);
(2)CREATE TABLE inhaber(name VARCHAR(20) NOT NULL, nummer INTEGER NOT NULL, PRIMARY
KEY(name, nummer));
(3)
(4)INSERT INTO konto VALUES(12345678, 5000);
(5)INSERT INTO konto VALUES(11111111, 100);
(6)
(7)INSERT INTO inhaber VALUES("John Doe", 12345678);
(8)INSERT INTO inhaber VALUES("John Doe", 11111111);
(9)INSERT INTO inhaber VALUES("Barnie Bar", 11111111);
  
```



[Download des Beispiels](#)

Das Transaktionsskript: Beispiel 46 zeigt die das Transaktionsskript realisierende Klasse. Sie kapselt die Datenbankinteraktion vollständig und stellt die beiden in [Abbildung 9](#) gezeigten Methoden zur Verfügung.

Beispiel 46: Transaktionsskript zur Interaktion mit der Bank


```

(1)import java.sql.Connection;
(2)import java.sql.DriverManager;
(3)import java.sql.ResultSet;
(4)import java.sql.SQLException;
(5)import java.sql.Statement;
(6)
(7)public class Bank {
(8)    public static void Überweisung(
(9)        int srcAccount,
(10)       int dstAccount,
(11)       double amount) {
(12)        Connection con = connectDB();
(13)        try {
(14)            con.setAutoCommit(false);
(15)            con.setTransactionIsolation(Connection.
TRANSACTION_SERIALIZABLE);
(16)            Statement stmt = con.createStatement();
(17)            stmt.executeUpdate("UPDATE konto set stand=stand-"+amount+"
where nummer='"+srcAccount+"'");
(18)            stmt.executeUpdate("UPDATE konto set stand=stand-"+amount+"
where nummer='"+dstAccount+"'");
(19)            con.commit();
(20)        } catch (SQLException e) {
(21)            e.printStackTrace();
(22)        }
(23)    }
(24)    public static String Vermögensübersicht(String kunde) {
(25)        String result=null;
(26)        try {
(27)            result= "Vermögensübersicht für: "+kunde+"\n";
(28)            result+="-----\n";
(29)            Statement stmt = connectDB().createStatement();
(30)            stmt.executeUpdate("LOCK TABLES konto READ, inhaber READ;");
(31)            ResultSet rs = stmt.executeQuery("SELECT k.nummer, stand
FROM konto AS k, inhaber AS i WHERE i.name='"+kunde+"' AND i.nummer=k.nummer;");
(32)            while(!rs.isLast()) {
(33)                rs.next();
(34)                result+=rs.getInt("nummer")+"\t"+rs.getInt("stand")
+"\n";
(35)            }
(36)            result+="-----\n";
(37)            rs = stmt.executeQuery("SELECT SUM(stand) AS s from konto as
k,inhaber as i where i.name='"+kunde+"' and i.nummer=k.nummer;");
(38)            stmt.executeUpdate("UNLOCK TABLES;");
(39)            rs.next();
(40)            result+="Saldo aller Konten: "+rs.getInt("s")+"\n";
(41)        } catch (SQLException e) {
(42)            e.printStackTrace();
(43)        }
(44)        return result;
(45)    }
(46)    private static Connection connectDB() {
(47)        try {
(48)            Class.forName("com.mysql.jdbc.Driver");
(49)        } catch (ClassNotFoundException e) {
(50)            System.err.println("Driver class not found");
(51)            e.printStackTrace();
(52)        }
(53)        Connection con = null;
(54)        try {
(55)            con =
(56)                (Connection) DriverManager.getConnection(
(57)                    "jdbc:mysql://localhost/bank/",
(58)                    "root",
(59)                    "");
(60)        } catch (SQLException e1) {
(61)            System.err.println("Error establishing database connection");
(62)            e1.printStackTrace();
(63)        }
(64)        return con;
(65)    }
(66)

```



`(67)}`[Download des Beispiels](#)

Die gesamte Interaktion mit der Persistenzlogik geschieht durch die beiden Geschäftsmethoden und bedarf keiner Kenntnis und Berücksichtigung der technischen Datenbankcharakteristika. Beispiel 47 zeigt die Nutzung der beiden angebotenen Methoden.

Beispiel 47: Nutzung des Transaktionsskriptes

```
(1)public class Driver {
(2)    public static void main(String args[]) {
(3)        Bank.Überweisung(12345678, 11111111, 500);
(4)        System.out.println(Bank.Vermögensübersicht("John Doe"));
(5)    }
(6)}
```

[Download des Beispiels](#)**Umsetzung unter Einsatz des *Command Musters*:**

Die Nutzung von Transaktionsskripten führt zur Bildung von Methoden, die mit Namen aus der Geschäftsdomäne belegt sind. So treten im Beispiel 46 die Methoden Überweisung und Vermögensübersicht auf.

Diese Benennungseigenschaft ist jedoch nicht immer gewünscht. Vielmehr strebt man bei der Implementierung häufig eine gleichartige Aufrufschnittstelle verschiedener Sachverhalte an. Daher findet sich häufig Transaktionsskripte mithilfe des *Command Musters* umgesetzt.

Dieses Muster definiert für alle aufrufbaren Domänenmethoden des Transaktionsskriptes eine einheitliche Schnittstelle (im Beispiel durch die Methode run verkörpert).

Der Einsatz des Musters hat jedoch keinen Einfluß auf die verwirklichte Domänenlogik, sondern ändert nur die Aufrufmimik.

Die Beispiele 48 mit 50 zeigen die modifizierte Umsetzung der einzelnen Methoden des Transaktionsskripts, die nun durch separate Klassen repräsentiert werden. Die Kontroll-Logik bleibt jedoch gegenüber der vorhergehenden Lösung unverändert.

Beispiel 48: TransactionScript.java

```
(1)public abstract class TransactionScript {
(2)    public void run(){
(3)    }
(4)}
```

[Download des Beispiels](#)**Beispiel 49: Überweisung.java**

```
(1)import java.sql.Connection;
(2)import java.sql.DriverManager;
(3)import java.sql.SQLException;
(4)import java.sql.Statement;
(5)
(6)public class Überweisung extends TransactionScript {
(7)    private int srcAccount;
(8)    private int dstAccount;
(9)    private double amount;
(10)
(11)    public Überweisung(int srcAccount, int dstAccount, double amount) {
(12)        this.srcAccount = srcAccount;
(13)        this.dstAccount = dstAccount;
(14)        this.amount = amount;
(15)    }
(16)    public void run() {
(17)        Connection con = connectDB();
(18)        try {
(19)            con.setAutoCommit(false);
(20)            con.setTransactionIsolation(Connection.
TRANSACTION_SERIALIZABLE);
(21)            Statement stmt = con.createStatement();
(22)            stmt.executeUpdate(
```



```

(23)                "UPDATE konto set stand=stand-"
(24)                + amount
(25)                + " where number='"
(26)                + srcAccount
(27)                + "';");
(28)            stmt.executeUpdate(
(29)                "UPDATE konto set stand=stand+"
(30)                + amount
(31)                + " where number='"
(32)                + dstAccount
(33)                + "';");
(34)            con.commit();
(35)        } catch (SQLException e) {
(36)            e.printStackTrace();
(37)        }
(38)    }
(39)    private static Connection connectDB() {
(40)        try {
(41)            Class.forName("com.mysql.jdbc.Driver");
(42)        } catch (ClassNotFoundException e) {
(43)            System.err.println("Driver class not found");
(44)            e.printStackTrace();
(45)        }
(46)        Connection con = null;
(47)        try {
(48)            con =
(49)                (Connection) DriverManager.getConnection(
(50)                    "jdbc:mysql://localhost/bank/",
(51)                    "root",
(52)                    "");
(53)        } catch (SQLException e1) {
(54)            System.err.println("Error establishing database connection");
(55)            e1.printStackTrace();
(56)        }
(57)        return con;
(58)    }
(59)}

```

[Download des Beispiels](#)

Beispiel 50: Vermögensübersicht.java

```

(1)import java.sql.Connection;
(2)import java.sql.DriverManager;
(3)import java.sql.ResultSet;
(4)import java.sql.SQLException;
(5)import java.sql.Statement;
(6)
(7)public class Vermögensübersicht extends TransactionScript {
(8)    private String kunde;
(9)    public String result;
(10)
(11)    public Vermögensübersicht(String kunde) {
(12)        this.kunde = kunde;
(13)    }
(14)
(15)    public void run() {
(16)        try {
(17)            result = "Vermögensübersicht für: " + kunde + "\n";
(18)            result += "-----\n";
(19)            Statement stmt = connectDB().createStatement();
(20)            ResultSet rs =
(21)                stmt.executeQuery(
(22)                    "SELECT k.nummer, stand FROM konto AS k,
(23)                    inhaber AS i WHERE i.name='"
(24)                    + kunde
(25)                    + "' AND i.nummer=k.nummer;");
(26)            while (!rs.isLast()) {
(27)                rs.next();
(28)                result += rs.getInt("nummer")

```



```

(29)                                     + rs.getInt("stand")
(30)                                     + "\n";
(31)                                     }
(32)             result += "-----\n";
(33)             rs =
(34)                 stmt.executeQuery(
(35)                     "SELECT SUM(stand) AS s from konto as k,
inhaber as i where i.name='"
(36)                                     + kunde
(37)                                     + "' and i.nummer=k.nummer;");
(38)             rs.next();
(39)             result += "Saldo aller Konten: " + rs.getInt("s") + "\n";
(40)         } catch (SQLException e) {
(41)             e.printStackTrace();
(42)         }
(43)     }
(44)     private static Connection connectDB() {
(45)         try {
(46)             Class.forName("com.mysql.jdbc.Driver");
(47)         } catch (ClassNotFoundException e) {
(48)             System.err.println("Driver class not found");
(49)             e.printStackTrace();
(50)         }
(51)         Connection con = null;
(52)         try {
(53)             con =
(54)                 (Connection) DriverManager.getConnection(
(55)                     "jdbc:mysql://localhost/bank/",
(56)                     "root",
(57)                     "");
(58)         } catch (SQLException e1) {
(59)             System.err.println("Error establishing database connection");
(60)             e1.printStackTrace();
(61)         }
(62)         return con;
(63)     }
(64)
(65)}

```

[Download des Beispiels](#)

Bei der Erstellung von Applikationen, welche die vereinheitlichten Schnittstellen nutzen zeigt sich das Resultat in Form einer gleichartigen Aufrufschnittstelle (im Beispiel die Methode run) für die verschiedenen Domänenmethoden:

Beispiel 51: Driver2.java



```

(1) public class Driver2 {
(2)     public static void main(String args[]) {
(3)         Überweisung ü = new Überweisung(12345678, 11111111, 500);
(4)         ü.run();
(5)
(6)         Vermögensübersicht v = new Vermögensübersicht("John Doe");
(7)         v.run();
(8)         System.out.println(v.result);
(9)     }
(10)}

```

[Download des Beispiels](#)

Abschließende Würdigung:

Das Transaktionsskript-Muster bietet eine vergleichsweise einfach nachvollziehbare Möglichkeit zur Entkopplung von Persistenz- und Geschäftslogik an. Jedoch tritt sehr schnell (wie in den Beispielen 49 und 50 anhand der Methode connectDB gezeigt) die Gefahr auf, daß gleichartiger Code in verschiedene Transaktionsskripte zu integrieren ist.

Überdies führen komplexe Geschäftslogiken, die sich partiell überlappen und gegenseitig enthalten zu aufwendigen Entwürfen in denen Coderedundanz nicht immer zu vermeiden ist.

Abhilfe kann hier die Verwendung eines *Domänenmodells* bieten.

Domain Model

Motivation und Grundidee: Hintergrund der Strukturform des *Domain Models* ist der Versuch die in der Analysephase vorgefundenen Objekte des betrachteten Problembereichs möglichst unverändert durch die Applikation zur Verfügung zu stellen und in die Datenbank zu übernehmen. Im Gegensatz zur Strukturierungsform des *Transaction Scripts* erfolgt der Anwendungsaufbau hierbei nicht an den Geschäftsprozessen orientiert, sondern rein Daten-getrieben.

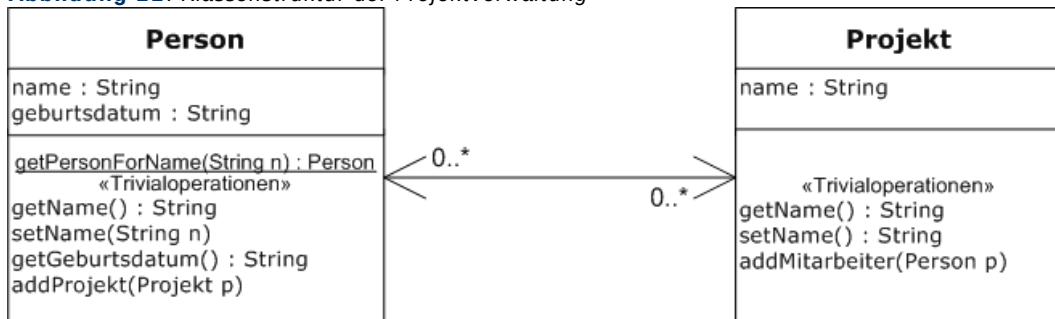
Struktur: Im Idealfall entsprechen sich die Struktur der applikationsimmanenten Klassen und die der datebankresidenten Tabellen eineindeutig. Abweichungen davon können sich lediglich durch Tabellen ergeben, welche dieselbe Realweltentität abbilden. Diese können durch den Normalisierungsprozeß gebildet worden sein.

Das Beispiel der [Abbildung 11](#) zeigt die Klassenstruktur einer Projektverwaltung, in der die Zuordnungen zwischen Personen und den Projekten in denen diese eingesetzt sind verwaltet werden.

Jeder Ausprägung von Person können hierbei mehrere Objekte des Typs Projekt zugeordnet sein und umgekehrt.

Zusätzlich sind die für die jeweiligen Klassen definierten Operation dargestellt. Hierbei kann es sich um triviale Operationen zum Setzen und Auslesen einzelner Attributwerte oder beliebig aufwendige Vorgänge handeln. Gemeinsames Kennzeichen aller Operationen ist jedoch, daß sie nur dasjenige Objekt betreffen in dessen Kontext sie definiert sind.

Abbildung 11: Klassenstruktur der Projektverwaltung

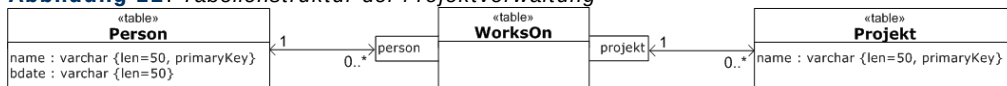


(click on image to enlarge!)

Die verwalteten Daten sind in drei Datenbanktabellen abgelegt. [Abbildung 12](#) zeigt die Struktur der Tabellen einschließlich der definierten Fremdschlüsselbeziehungen.

Aus Gründen der Normalisierung (die Relation befindet sich in [vierter Normalform](#)) wird zusätzlich die Tabelle WorksOn eingeführt, welche die Daten über die Zuordnung zwischen Projekten und den sie bearbeitenden Personen enthält.

Abbildung 12: Tabellenstruktur der Projektverwaltung



(click on image to enlarge!)

Der Aufbau der Datenbank geschieht durch das folgende SQL-Script:

Beispiel 52: DB-Aufbau

```

(1)CREATE TABLE Person(
(2)    name VARCHAR(50) PRIMARY KEY,
(3)    bdate DATE);
(4)
(5)CREATE TABLE Projekt(
(6)    name VARCHAR(50) PRIMARY KEY);
(7)
(8)CREATE TABLE WorksOn(
(9)    person VARCHAR(50) NOT NULL,
(10)   projekt VARCHAR(50) NOT NULL,
(11)   PRIMARY KEY (person, projekt));
(12)
(13)ALTER TABLE WorksOn ADD INDEX WO_person_IDX(person);
(14)ALTER TABLE WorksOn ADD CONSTRAINT WO_person_FK FOREIGN KEY (person) REFERENCES
Person(name);
(15)ALTER TABLE WorksOn ADD INDEX WO_project_IDX(projekt);
(16)ALTER TABLE WorksOn ADD CONSTRAINT WO_project_FK FOREIGN KEY (projekt)
REFERENCES Projekt(name);
  
```



[Download des Beispiels](#)

Das Domänenmodell: Die Beispiele 53 und 54 zeigen die beiden fachlichen Domänenklassen Person und Projekt.

Beispiel 53: Die Domänenklasse Person

```
(1)import java.sql.ResultSet;
(2)import java.sql.SQLException;
(3)import java.sql.Statement;
(4)import java.util.Vector;
(5)
(6)public class Person {
(7)    private String name;
(8)    private Vector projekte = new Vector();
(9)
(10)    public void setName(String name) {
(11)        Statement stmt = DBConnector.getConnectionStatement();
(12)        try {
(13)            ResultSet rs =
(14)                stmt.executeQuery(
(15)                    "SELECT count(*) FROM Person WHERE name='"
(16)                        + this.name
(17)                        + "';");
(18)            rs.next();
(19)            if (rs.getDouble(1) < 1) {
(20)                System.out.println(
(21)                    "Database entry for person does not exist
and will be created");
(22)                stmt.executeUpdate(
(23)                    "INSERT INTO Person (name) VALUES('" + name
+ "')");
(24)            } else {
(25)                System.out.println(
(26)                    "Database entry for person exists and will
be updated");
(27)                stmt.executeUpdate(
(28)                    "UPDATE Person SET name='"
(29)                        + name
(30)                        + "' where name='"
(31)                        + this.name
(32)                        + "';");
(33)            }
(34)        } catch (SQLException e) {
(35)            System.out.println("Cannot access database");
(36)            e.printStackTrace();
(37)        }
(38)        this.name = name;
(39)    }
(40)    public void setGeburtsdatum(String gebDat) {
(41)        Statement stmt = DBConnector.getConnectionStatement();
(42)        try {
(43)            System.out.println("Database entry (bdate) will be updated");
(44)            stmt.executeUpdate(
(45)                "UPDATE Person SET bdate='"
(46)                    + gebDat
(47)                    + "' WHERE name='"
(48)                    + this.name
(49)                    + "';");
(50)        } catch (SQLException e) {
(51)            System.out.println("Cannot access database");
(52)            e.printStackTrace();
(53)        }
(54)    }
(55)    public String getGeburtsdatum() {
(56)        Statement stmt = DBConnector.getConnectionStatement();
(57)        ResultSet rs;
(58)        String result = "";
(59)        try {
(60)            rs =
(61)                stmt.executeQuery(
(62)                    "SELECT bdate FROM Person WHERE name='" +
name + "';");
(63)            rs.next();
```



```

(64)                result = rs.getString(1);
(65)            } catch (SQLException e) {
(66)                // TODO Auto-generated catch block
(67)                e.printStackTrace();
(68)            }
(69)            return result;
(70)        }
(71)        public String getName() {
(72)            return name;
(73)        }
(74)        public void addProjekt(Projekt p) {
(75)            Statement stmt = DBConnector.getConnectionStatement();
(76)            try {
(77)                stmt.executeUpdate(
(78)                    "INSERT INTO WorksOn VALUES('"
(79)                        + this.name
(80)                        + "',''"
(81)                        + p.getName()
(82)                        + "');");
(83)            } catch (SQLException e) {
(84)                System.out.println("Cannot access database");
(85)                e.printStackTrace();
(86)            }
(87)        }
(88)        public Vector getAllProjekt() {
(89)            Vector projekte = new Vector();
(90)
(91)            return projekte;
(92)        }
(93)        public static Person getPersonForName(String name) {
(94)            Statement stmt = DBConnector.getConnectionStatement();
(95)            ResultSet rs;
(96)            Person p = new Person();
(97)            try {
(98)                rs =
(99)                    stmt.executeQuery(
(100)                        "SELECT bdate FROM Person WHERE name='" +
name + "'");
(101)                rs.next();
(102)                p.name = name;
(103)            } catch (SQLException e) {
(104)                System.out.println("Cannot access database");
(105)                e.printStackTrace();
(106)            }
(107)            return p;
(108)        }
(109)        public String toString() {
(110)            return "(name: "+name+", bdate: "+this.getGeburtsdatum()+")";
(111)        }
(112)    }

```

[Download des Beispiels](#)

Beispiel 54: Die Domänenklasse Projekt

```

(1)import java.sql.ResultSet;
(2)import java.sql.SQLException;
(3)import java.sql.Statement;
(4)import java.util.Vector;
(5)
(6)public class Projekt {
(7)    private String name;
(8)    private Vector mitarbeiter = new Vector();
(9)
(10)    public void setName(String name) {
(11)        Statement stmt = DBConnector.getConnectionStatement();
(12)        try {
(13)            ResultSet rs =
(14)                stmt.executeQuery(
(15)                    "SELECT count(*) FROM Project WHERE name='"
+ this.name

```



```

(17)                                + "';");
(18)                                rs.next();
(19)                                if (rs.getDouble(1) < 1) {
(20)                                    System.out.println(
(21)                                        "Database entry for project does not exist
and will be created");
(22)                                    stmt.executeUpdate(
(23)                                        "INSERT INTO Project (name) VALUES('" + name
+ "')");
(24)                                } else {
(25)                                    System.out.println(
(26)                                        "Database entry for project exists and will
be updated");
(27)                                    stmt.executeUpdate(
(28)                                        "UPDATE Project SET name='"
(29)                                            + name
(30)                                            + "' where name='"
(31)                                            + this.name
(32)                                            + "';");
(33)                                }
(34)                                } catch (SQLException e) {
(35)                                    System.out.println("Cannot access database");
(36)                                    e.printStackTrace();
(37)                                }
(38)                                this.name = name;
(39)                            }
(40)                            public String getName() {
(41)                                return name;
(42)                            }
(43)                            public void addMitarbeiter(Person p) {
(44)                                Statement stmt = DBConnector.getConnectionStatement();
(45)                                try {
(46)                                    stmt.executeUpdate(
(47)                                        "INSERT INTO WorksOn VALUES('"
(48)                                            + p.getName()
(49)                                            + "',''"
(50)                                            + this.name
(51)                                            + "');");
(52)                                } catch (SQLException e) {
(53)                                    System.out.println("Cannot access database");
(54)                                    e.printStackTrace();
(55)                                }
(56)                            }
(57)                            public Vector getAllMitarbeiter() {
(58)                                Vector mitarbeiter = new Vector();
(59)                                Statement stmt = DBConnector.getConnectionStatement();
(60)                                Person p;
(61)                                try {
(62)                                    ResultSet rs = stmt.executeQuery("SELECT name FROM Person as
p, WorksOn as w WHERE w.project='"+this.name+"' and w.person=p.name;");
(63)                                    while (!rs.isLast()) {
(64)                                        rs.next();
(65)                                        p = Person.getPersonForName(rs.getString(1));
(66)                                        mitarbeiter.add(p);
(67)                                    }
(68)                                } catch (SQLException e) {
(69)                                    System.out.println("Cannot access database");
(70)                                    e.printStackTrace();
(71)                                }
(72)                                return mitarbeiter;
(73)                            }
(74)
(75)}

```



[Download des Beispiels](#)

Zusätzlich ist aus Gründen der vereinfachten Interaktion mit dem Datenbankmanagementsystem die Klasse DBConnector umgesetzt.

Beispiel 55: Die Klasse DBConnector

```

(1)import java.sql.Connection;
(2)import java.sql.DriverManager;
(3)import java.sql.SQLException;
(4)import java.sql.Statement;
(5)
(6)public class DBConnector {
(7)    public static Statement getConnectedStatement() {
(8)        try {
(9)            Class.forName("com.mysql.jdbc.Driver");
(10)        } catch (ClassNotFoundException e) {
(11)            System.err.println("Driver class not found");
(12)            e.printStackTrace();
(13)        }
(14)        Connection con = null;
(15)
(16)        try {
(17)            con =
(18)                (Connection) DriverManager.getConnection(
(19)                    "jdbc:mysql://dbServerMachine/mcjttest/",
(20)                    "user",
(21)                    "thePassword");
(22)            return (con.createStatement());
(23)        } catch (SQLException e1) {
(24)            System.err.println("Error establishing database connection");
(25)            e1.printStackTrace();
(26)        }
(27)        //never gets here
(28)        return null;
(29)    }
(30)}

```



[Download des Beispiels](#)

Die beiden Domänenklassen kapseln die Interaktion mit der Datenbank vollständig vor dem Aufrufer. Alle Persistenzoperationen werden in Form einfacher („low-level“) Operationen zur Verfügung gestellt. Die durch eine Klasse angesprochenen Datenbanktabellen sind dabei streng auf diejenigen beschränkt, welche die durch die Klasse verwaltenden Daten aufnehmen.

Auffallend ist, daß die Domänenobjekte außer den Attributen, die den Primärschlüssel repräsentieren, keine durch Attribute ausgedrückte Eigenschaften besitzen. Diese werden ausschließlich in der Datenbank repräsentiert und im Bedarfsfalle angefragt (Beispiel: Realisierung der Methode `getGeburtsdatum` der Klasse `Person`).

Die Begründung hierfür wird bei der Analyse der Zugriffe auf die Tabelle `WorksOn` offenkundig. Da diese Tabelle durch die beiden Domänenobjekte unabhängig voneinander zugegriffen werden kann, würde eine (redundante) Datenverwaltung im Hauptspeicher tendenziell zu Konsistenzproblemen mit durch das Datenbankmanagementsystem verwalteten Daten führen.

Einen solchen Fall, der durch die gewählte Umsetzung korrekt behandelt wird, zeigt der in Beispiel 56 wiedergegebene Code:

Beispiel 56: Domänenmodell verwendende Applikation

```

(1)public class TestDriver {
(2)    public static void main(String args[]) {
(3)        Person pers1 = new Person();
(4)        pers1.setName("Max Mustermann");
(5)        pers1.setName("XX");
(6)        pers1.setName("Max Mustermann");
(7)        pers1.setGeburtsdatum("1970-11-12");
(8)
(9)        Person pers2 = new Person();
(10)       pers2.setName("John Doe");
(11)
(12)       Projekt prj1 = new Projekt();
(13)       prj1.setName("Reorganisation");
(14)
(15)       pers1.addProjekt(prj1);
(16)       prj1.addMitarbeiter(pers2);
(17)
(18)       System.out.println(prj1.getAllMitarbeiter());
(19)    }
(20)

```



(21)}

Download des Beispiels

Im Beispiel werden zunächst Ausprägungen des Typs Person und Projekt erzeugt und manipuliert. Hierbei wird für dasselbe Personen-Objekt eine Zuordnung dieses Objekts zu einem Projekt vorgenommen und anschließend diesem Projekt eine andere Person zugeordnet. Diese beiden Interaktionen entsprechen der Instanziierung der die beiden Domänenklassen verbindenden Assoziation mit jeweils unterschiedlichen Ausgangspunkten der Beziehungsetablierung. Nur durch den Rückgriff auf die Datenbankinhalte liefert der Aufruf von `getAllMitarbeiter` konsistente Daten, da sowohl innerhalb der Person- als auch der Projekt-Ausprägung nur unvollständige (d.h. diejenigen durch den jeweiligen `add...`-Aufruf erzeugten) Daten vorliegen.

Abschließende Würdigung: Das Domänenmodell ermöglicht eine vergleichsweise einfache Abbildung der Objektstrukturen des Hauptspeichers in relationale Datenbankstrukturen. Allerdings ist die Abbildung komplexer Abhängigkeitsstrukturen der objektorientierten Applikationsdaten in relationale Tabellenstrukturen mitunter schwierig; insbesondere wenn hinsichtlich der Güte der entstehenden DB-Strukturen zusätzliche Qualitätskriterien (wie Redundanzfreiheit durch Normalisierung) angelegt werden.

Grundsätzlich bietet dieses Umsetzungsmuster den Vorteil aus Sicht des Fachanwenders „naheliegende“ Entitäten bereitstellen zu können.

Die Interaktion auf der Basis der durch Domänenklassen angebotenen Operationen kann, abhängig vom Komplexitätsgrad der zu realisierenden Anwendung, -- im Vergleich zum Ansatz des *Transaction Scripts* -- aufwendig sein. Insbesondere offenbart sich die vermöge der Domänenoperationen etablierte Abstraktionsschicht als Hemmnis, wenn eine direkte Interaktion mit den wertrepräsentierenden Tabellen intendiert ist.

In diesen Fällen eignet sich ein *Table Module* besser zur Realisierung.

Service provided by [Mario Jeckle](#)

Generated: 2004-01-08T22:56:54+01:00

► [Feedback](#)

► [SiteMap](#)

► [This page's original location: http://www.jeckle.de/vorlesung/db-anwendungen/script.html](http://www.jeckle.de/vorlesung/db-anwendungen/script.html)

► [RDF description for this page](#)

▲ Vorlesungen

▼ Vorlesungsinhalt

▼ Literatur

▼ Inhaltsübersicht und Zugang zum Script

▼ Download des Scripts

▼ Links

▼ Mailingliste zur Vorlesung

▶ Aktuelles **XML**

▲ Organisatorisches

Die Vorlesung findet donnerstags von 7.45 bis 11 Uhr im Raum A314 statt.
Praktikum wird donnerstags von 14 bis 15.30 Uhr im Raum C103 angeboten.

▲ Abstract

Die Vorlesung vermittelt, aufbauend auf der Grundvorlesung, einen Überblick aktueller Anwendungsaspekte von Datenbanken.

Durchgängiger Schwerpunkt der Veranstaltung ist das Verständnis für geschichtete Architekturen, ihre Vorteile und die zur Realisierung eingesetzten Techniken.

Hierzu werden zunächst einführend verschiedene Zugriffsschnittstellen diskutiert, die sich von SQL-basierten Zugriffen in Programmiersprachen über weiter abstrahierte Programmierschnittstellen bis hin zu transparenten Persistenzansätzen spannen. Die Betrachtung fokussiert auf die Java-Techniken der Standard- und Enterprise-Plattform.

Aufbauend auf den Schnittstellentechniken werden Aspekte der internen Realisierung einzelner Architekturschichten vorgestellt. Im Vordergrund der Betrachtung steht dabei die Identifikation wiederverwendbarer Lösungsansätze, die realisierungsunabhängig und daher auf verschiedene Problemstellungen übertragbar sind.

Abschließend werden Techniken zum Zugriff auf datenbankbasierte Applikationen vorgestellt. Dabei wird mit der Extensible Markup Language und den darauf basierenden Techniken ein Ansatz zur plattformneutralen Speicherung in den Vordergrund gestellt. Ausgehend von dieser Betrachtung wird mit Web Services eine Möglichkeit zur Umsetzung einer systemunabhängigen Zugriffsschnittstelle auf Daten diskutiert.

Inhaltsübersicht

- 1 [Zugriffsschnittstellen](#)
 - 1.1 [Java Database Connectivity](#)
 - 1.2 [Enterprise Java Beans](#)
 - 1.3 [Java Data Objects](#)
- 2 [Architekturmuster und Umsetzungstechniken](#)
 - 2.1 [Domänenlogik](#)
 - 2.1.1 [Transaction Script](#)
 - 2.1.2 [Domain Model](#)
 - 2.1.3 [Table Module](#)
 - 2.2 [Datenzugriff](#)
 - 2.2.1 [Table Data Gateway](#)
 - 2.2.2 [Row Data Gateway](#)
 - 2.2.3 [Active Record](#)
 - 2.3 [Objekt-Relational-Abbildung und -Interoperabilität](#)
 - 2.3.1 [ID-Feld](#)
 - 2.3.2 [Identity Map](#)
 - 2.3.3 [Lazy Load](#)
- 3 [Konnektivität und Offline-Techniken](#)
 - 3.1 [XML-Strukturen](#)
 - 3.2 [XML-Schema](#)
 - 3.3 [XSL-Transformationen](#)
 - 3.4 [XML-Programmierschnittstellen](#)
 - 3.5 [XML und Datenbanken](#)
 - 3.6 [Web Services](#)

Download des Scripts

- [Scriptum \(tief verlinkt\) als PDF-Datei](#)
- [Scriptum als PDF-Datei](#)
- [Scriptum als XML-Datei](#)
- [Beispiele des Scriptums als TGZ-Archiv](#)
- [Software für das Praktikum \(85 MB!\)](#)

Links

Wird noch ergänzt ...

SQLJ

- [SQLJ: The 'open sesame' of Java database applications](#)
- [SQLJ.org](#)
- [SQLJ @ O'Reilly](#)
- [SQLJ @ OnJava.com](#)
- [SQLJ @ Oracle](#)
- [SQLJ vs. JDBC](#)

Literatur

Wird noch ergänzt ...

Java Data Objects

- D. Jordan, C. Russell: [*Java Data Objects*](#), O'Reilly, 2003.
- R. M. Ross: [*Java Data Objects*](#), Addison Wesley, 2003.
- S. Tyagi, M. Vorburger, K. McCammon, H. Bobzin, K. McCannon: [*Core Java Data Objects*](#), Prentice Hall, 2003.

Java Database Connectivity

- G. Reese: [*Database Programming with JDBC and Java*](#), O'Reilly, 1997

Architekturmuster und Umsetzungstechniken

- M. Fowler: *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.
Deutsche Übersetzung: *Patterns für Enterprise Application-Architekturen*, MITP, 2003.

Mailingliste

Steht im [Wintra-System](#) zur Verfügung.

Service provided by [Mario Jeckle](#)

Generated: 2003-12-24T12:59:07+01:00

 [Feedback](#)  [SiteMap](#)

 [This page's original location: http://www.jeckle.de/vorlesung/db-anwendungen/index.html](#)

 [RDF description for this page](#)



[Home](#) | [Site map](#) | [Abbreviations](#) | [ISO Store](#)
[Français](#) | [FAQ](#) | [Contact ISO](#) | [My account](#)
Search Extended Search

[About ISO](#) | [Products and services](#) | [ISO 9000 / 14000](#) | [Standards development](#) | [Communities and markets](#) | [Communication centre](#)



**ISO
Catalogue**

ICS
fields

35

Information technology.
Office machines

35.060

**Languages used in
information technology**



[View Shopping Basket](#)

The URL you've requested doesn't link to a valid catalogue entry.

Search options

- ☒ Text
☐ ISO Number

Type in search string

[Help on using search](#)

[Extended Search](#)

- [How to use the catalogue](#)
- [Maintenance agencies and
Registration Authorities](#)
- [List of withdrawn standards](#)

©
ISO

ISO name and
logo

Privacy
policy



The Source for Java Developers

[Developers Home](#) > [Products & Technologies](#) > [Java Technology](#) > [J2EE](#) > [JDBC](#) >[Profile and Registration](#) | [Why Register?](#)

JDBC Downloads

Downloads

- [Early Access](#)
- [Tools](#)

[Japanese](#)
[日本語版](#)

Reference

- [API Specifications](#)
- [Documentation](#)
- [FAQs](#)
- [Code Samples and Apps](#)
- [Technical Articles and Tips](#)
- [Industry Support](#)

Community

- [Bookshelf](#)
- [Code Certification](#)
- [Bug Database](#)
- [Forums](#)

Learning

- [Tutorial and Code Camps](#)
- [Online Courses](#)
- [Instructor-Led Courses](#)
- [Quizzes](#)

This page provides the following downloads and specifications:

- [JDBC Rowset Implementation 1.0 Proposed Final Draft](#)
- [JDBC Rowset Implementation 1.0 Public Review Draft](#)
- [JDBC 3.0 API](#)
- [JDBC Optional Package for CDC/Foundation Profile API](#)
- [JDBC 2.1 API](#)
- [JDBC 2.0 Optional Package API](#)
- [JDBC API Test Suite 1.3.1](#)
- [JDBC API Test Suite 1.2.1](#)

JDBC RowSet Implementations

The JDBC RowSet Implementations specification standardizes key RowSet implementations defined in the JDBC 3.0 specification. This specification will become part of the Java 2 Standard Edition 1.5 (Tiger) when it is released.

JDBC 3.0 API

All of the JDBC 3.0 API, including both the `java.sql` and `javax.sql` packages, is bundled with the Java 2 Platform, Standard Edition, version 1.4 (J2SE). If you have downloaded the J2SE platform, you already have all of the JDBC API and do not need to download anything from this page.

JDBC Optional Package for CDC/Foundation Profile API

If you write applications for the Java 2 Platform, Micro Edition, J2ME that access a relational database, you need to download this Optional Package. It is a subset of the `java.sql` package with a smaller footprint that is tailored to writing applications running CDC (Connected Device Configuration).

JDBC 2.1 Core API and JDBC 2.0 Optional Package API

General Application Programmers

If you write only client-side applications, you do not need to download anything from this page. By downloading the Java 2 SDK, Standard Edition, you already have the JDBC core API (`java.sql` package) and

Related Links

Popular Downloads

- [Java Web Services Developer Pack 1.3](#)
- [J2EE 1.4 SDK](#)
- [J2SE 1.4.2](#)
- [Sun Java System Application Server](#)

Products and Technologies

- [JDBC Drivers](#)
- [Java Data Objects](#)
- [JDBC-ODBC Bridge](#)

Sun Resources

- [New to Java Center](#)
- [Developer Technical Support](#)
- [JavaOne Online](#)
- [Professional Training](#)
- [Professional Certification](#)

the associated javadoc documentation.

Server-side Programmers

If you write Enterprise JavaBeans (EJB) components, any software that supports them, or any other server-side software, you need to download the Java 2 SDK, Enterprise Edition. The Enterprise Edition includes both the JDBC core API and the JDBC Optional Package API (the `javax.sql` package). Both packages are necessary for any JDBC application that uses a `DataSource` object to make a connection, uses connection pooling, or uses distributed transactions.

Client-side Programmers Using a `DataSource` or `RowSet` Object

In general, you might want to download the `javax.sql` package in addition to the Java 2 SDK, Standard Edition, if you do not write server-side code but want to use one of the following:

- A `DataSource` object to make a connection
Note: To use a `DataSource` object to establish a connection, which is the recommended way, your driver must include a basic `DataSource` implementation.
- A `RowSet` object
 Rowsets are typically used as a convenient way to pass data or to add scrollability to a result set.

Downloading the `javax.sql` package instead of the entire Enterprise Edition will save you disc space. If your driver vendor bundles the `javax.sql` package with its product, as many are expected to do, you do not need to download anything beyond the Java 2 SDK, Standard Edition.

NOTE: The `javax.sql` package is also called the JDBC 2.0 Optional Package API (formerly known as the JDBC 2.0 Standard Extension API).

DOWNLOADS

JDBC API SPECIFICATIONS
The JDBC API Specification was created under the Java Community Process to provide full public participation in the definition and development.
JDBC Rowset Implementations 1.0 - Public Review Draft 2 NEW (Dec 12, 2003)

Specification	JDBC Rowset Implementations 1.0 Specification Proposed Final Draft
Send comments to	jsr-114-comments@jcp.org
JDBC Rowset Implementations 1.0 - Public Review Draft 2 (<i>Aug 28, 2003</i>)	
Specification	JDBC Rowset Implementations 1.0 Specification Public Review 2
Reference Implementation	JDBC Rowset Implementations 1.0 Reference Implementation Public Review 2
Send comments to	jsr-114-comments@jcp.org
JDBC Rowset Implementations 1.0 - Public Review Draft (<i>June 26, 2003</i>)	
Specification	JDBC Rowset Implementations 1.0 Public Review Specification
Send comments to	jsr-114-comments@jcp.org
JDBC 3.0 - Final Release (<i>February 13, 2002</i>)	
Specification	This Specification is available for viewing on-line or by downloading the document. Any use or implementation of this Specification is subject to this License agreement. Select a format Continue The Adobe Acrobat Reader software is required to view the PDF format of this specification. Adobe Acrobat Reader is free and available from the Adobe Systems website.
Send comments to	jdbc@sun.com

JDBC Optional Package for CDC/Foundation Profile - Proposed Final Draft (November 5, 2002)	
Specification	This Specification is available for viewing on-line or by downloading the document. Any use or implementation of this Specification is subject to this License agreement. Continue
Send comments to	jdbc-cdc@sun.com
JDBC Optional Package for CDC/Foundation Profile - Public Draft (August 17, 2002)	
Specification	This Specification is available for viewing on-line or by downloading the document. Any use or implementation of this Specification is subject to this License agreement. Continue
Send comments to	jdbc-cdc@sun.com
JDBC 2.1	
JDBC 2.1 Core API	Select a format continue The Adobe Acrobat Reader software is required to view the PDF format of this specification. Adobe Acrobat Reader is free and available from the Adobe Systems website.
JDBC 2.1 Specification	View 2.1 Errata JCP 2.0 Maintenance Review APPROVED
Java 2 Platform Documentation	Java 2 Platform Documentation (contains the JDBC 2.1 Core API javadoc)
JDBC 2.1 Core Source	JDBC 2.1 Core API Source is part of Java 2
JDBC 2.1 Core Binary	JDBC 2.1 Core API is part of Java 2
JDBC 2.0 Optional Package	

JDBC 2.0 Optional package API	 PostScript format (430367 bytes)  PDF format (327005 bytes) The Adobe Acrobat Reader software is required to view the PDF format of this specification. Adobe Acrobat Reader is free and available from the Adobe Systems website.
JDBC 2.0 Optional Package documentation	Download the JDBC 2.0 Optional Package API documentation: continue Browse the JDBC 2.0 Optional Package API documentation online. Download the JDBC 2.0 Optional Package API documentation:
JDBC 2.0 Optional Package Source	This download contains the .java files for the javax.sql package. These consist mainly of interfaces, and a few small helper classes. The javax.sql package does not contain a JDBC driver, and it does not contain a JDBC rowset implementation. continue
JDBC 2.0 Optional Package Binary	This download contains a compiled version (. class files) of the javax. sql package. It is appropriate for inclusion in a classpath, and may be added to a Java 2 installation. continue
JDBC API Test Suite 1.3.1 (<i>February 21, 2002</i>)	
JDBC API Test Suite 1.3.1 Documentation	JDBC API Test Suite 1.3.1 Documentation for Installation and Execution

JDBC API Test Suite 1.3.1	Continue
JDBC API Test Suite 1.3.1. Exclude List	Download
JDBC API Test Suite 1.2.1	
JDBC API Test Suite 1.2.1 Documentation	JDBC API Test Suite 1.2.1 Documentation for Installation and Execution
JDBC API Test Suite 1.2.1	Continue
JDBC API Test Suite 1.2.1. Exclude List	Download

Looking for JDBC 1.x API Downloads? Find them [here](#).



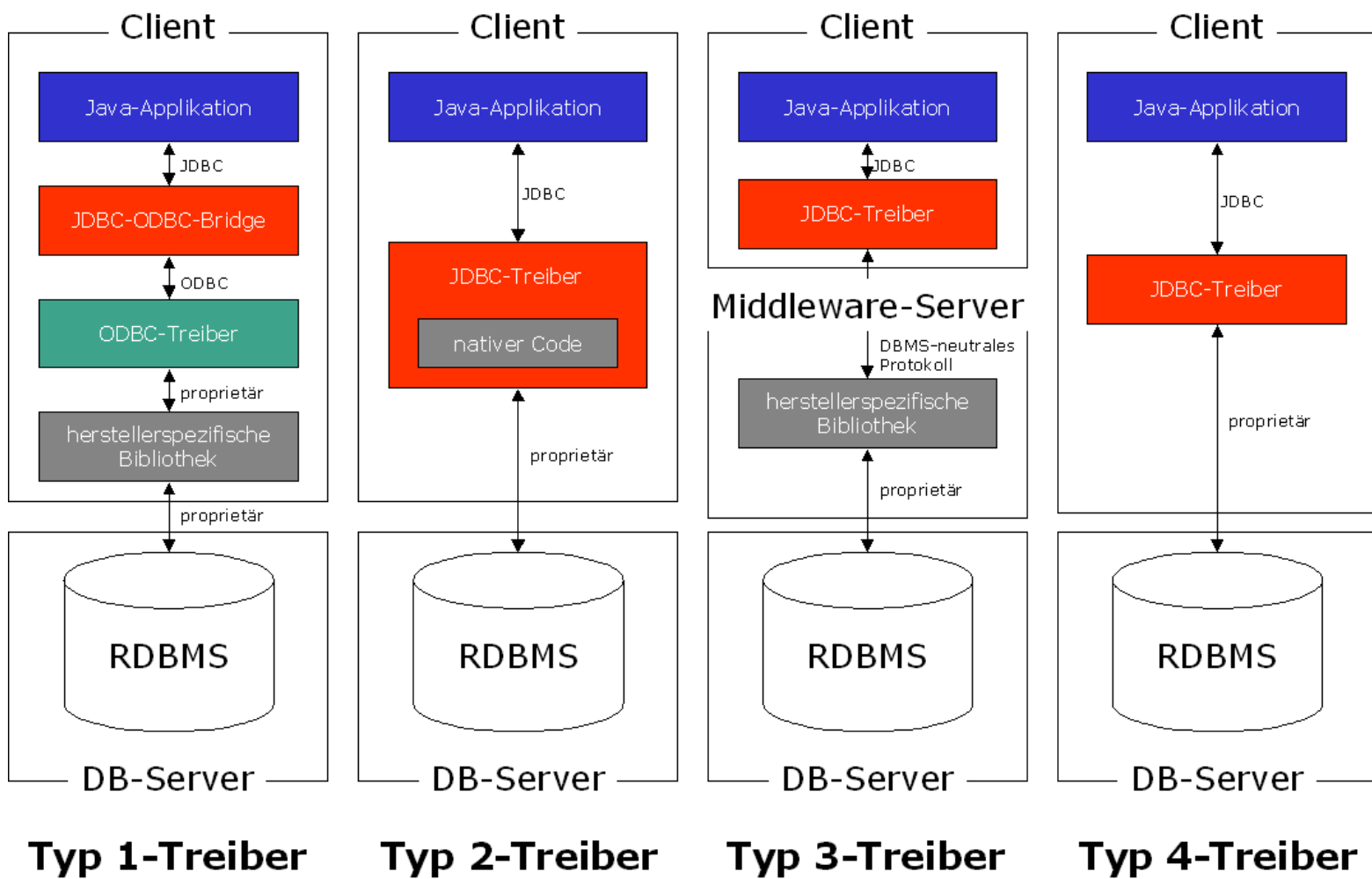
[Company Info](#) | [About SDN](#) | [Press](#) | [Contact Us](#) |
[Employment](#)
[How to Buy](#) | [Licensing](#) | [Terms of Use](#) | [Privacy](#) |
[Trademarks](#)

Copyright 1994-2004 Sun Microsystems, Inc.

A Sun Developer Network Site

Unless otherwise licensed, code in all technical manuals herein (including articles, FAQs, samples) is provided under this [License](#).

[XML](#) [Content Feeds](#)





MySQL® Connector/J

MySQL Connector/J is a native Java driver that converts JDBC (Java Database Connectivity) calls into the network protocol used by the MySQL database. It lets developers working with the Java programming language easily build programs and applets that interact with MySQL and connect all corporate data, even in a heterogeneous environment. MySQL Connector/J is a Type IV JDBC driver and has a complete JDBC feature set that supports the capabilities of MySQL.

Features

The latest production version of the driver is now 50-100 percent faster in most situations than the previous version. It also creates fewer transient objects than before, leading to better performance and even more stability. The driver now also supports "streaming" result sets, which allows users to retrieve large numbers of rows without using a large memory buffer. With newly added large-packet protocol support, the driver can send rows and BLOBs up to 2 gigabytes in size.

New features from the JDBC-3.0 API in the latest production version of MySQL Connector/J include `getGeneratedKeys` which allows users to retrieve auto-increment fields in a non-database-specific way. Auto-increment fields now work with object-relational mapping tools, as well as Enterprise Java Beans (EJB) servers with Container Managed Persistence (CMP) that support JDBC-3.0.

The development version is being refactored to support new features in conjunction with version 4.1 of the MySQL database server, including server-side prepared statements and improved handling of multiple simultaneous character sets, including Unicode in the UCS2 and UTF8 encodings.

Downloads, Licensing, and Support

Three versions of MySQL Connector/J are available for download:

- [MySQL Connector/J 3.0](#) is the production-ready version of the driver, and is available

"I highly recommend it."

anyone. [MySQL Connector/J 3.1](#) is the development version of the driver, and is available under purchase their support.

Anytime you've had to email [MySQL Connector/J 2.0](#) is the old production-ready version of the driver, and is available under the LGPL.

question, within minutes we

have had an answer back."

Commercial licenses for either version can also be purchased from MySQL AB, for those who don't wish to be bound by the LGPL or GPL. For more information on licensing MySQL

Connector/J, please [contact us](#). MySQL AB also offers [commercial support for MySQL](#)

Connector/J.

[Learn more about MySQL](#)

[support....](#)

Interoperability

Related pages: MySQL Connector/J is known to work with the following database tools and application servers:

- [Apache's Jakarta Projects](#) (Tomcat, Turbine, Velocity, etc.) — MySQL Connector/J works with all of these tools, and in most cases is the JDBC driver used in the example applications provided.
- [JBoss](#) The JBoss Open Source J2EE Application server. MySQL Connector/J works with both BMP and CMP beans in this environment .
- [BEA WebLogic](#) — MySQL Connector/J works with both BMP and CMP beans as well as servlets and JSPs on this commercial J2EE platform.
- [IBM VisualAge for Java](#) — VisualAge is a Java development environment.
- [IBM WebSphere Application Server 4.0](#) — MySQL Connector/J works with this commercial J2EE platform. BMP, servlets, and JSPs are supported.
- [NetBeans](#) — its open source counterpart, [NetBeans](#) — A world-class, professional IDE is a platform plus modules that includes that include things such as a code editor, tools for working with source code (Java, C++, and others), version control, and a lot more.
- [TableGen](#) — TableGen automatically generates classes to represent tables within a database. It is released under the GPL.

[java.sql](#)

Interfaces

[Array](#)

[Blob](#)

[CallableStatement](#)

[Clob](#)

[Connection](#)

[DatabaseMetaData](#)

[Driver](#)

[ParameterMetaData](#)

[PreparedStatement](#)

[Ref](#)

[ResultSet](#)

[ResultSetMetaData](#)

[Savepoint](#)

[SQLData](#)

[SQLInput](#)

[SQLOutput](#)

[Statement](#)

[Struct](#)

Classes

[Date](#)

[DriverManager](#)

[DriverPropertyInfo](#)

[SQLPermission](#)

[Time](#)

[Timestamp](#)

[Types](#)

Exceptions

[BatchUpdateException](#)

[DataTruncation](#)

[SQLException](#)

[SQLWarning](#)

java.sql

Interface Connection

public interface **Connection**

A connection (session) with a specific database. SQL statements are executed and results are returned within the context of a connection.

A `Connection` object's database is able to provide information describing its tables, its supported SQL grammar, its stored procedures, the capabilities of this connection, and so on. This information is obtained with the `getMetaData` method.

Note: By default a `Connection` object is in auto-commit mode, which means that it automatically commits changes after executing each statement. If auto-commit mode has been disabled, the method `commit` must be called explicitly in order to commit changes; otherwise, database changes will not be saved.

A new `Connection` object created using the JDBC 2.1 core API has an initially empty type map associated with it. A user may enter a custom mapping for a UDT in this type map. When a UDT is retrieved from a data source with the method `ResultSet.getObject`, the `getObject` method will check the connection's type map to see if there is an entry for that UDT. If so, the `getObject` method will map the UDT to the class indicated. If there is no entry, the UDT will be mapped using the standard mapping.

A user may create a new type map, which is a `java.util.Map` object, make an entry in it, and pass it to the `java.sql` methods that can perform custom mapping. In this case, the method will use the given type map instead of the one associated with the connection.

For example, the following code fragment specifies that the SQL type `ATHLETES` will be mapped to the class `Athletes` in the Java programming language. The code fragment retrieves the type map for the `Connection` object `con`, inserts the entry into it, and then sets the type map with the new entry as the connection's type map.

```
java.util.Map map = con.getTypeMap();
map.put("mySchemaName.ATHLETES", Class.forName("Athletes"));
con.setTypeMap(map);
```

See Also:

[DriverManager.getConnection\(java.lang.String, java.util.Properties\)](#), [Statement](#), [ResultSet](#), [DatabaseMetaData](#)

Field Summary

static int	TRANSACTION_NONE A constant indicating that transactions are not supported.
static int	TRANSACTION_READ_COMMITTED A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
static int	TRANSACTION_READ_UNCOMMITTED A constant indicating that dirty reads, non-repeatable reads and phantom reads can occur.
static int	TRANSACTION_REPEATABLE_READ A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur.
static int	TRANSACTION_SERIALIZABLE A constant indicating that dirty reads, non-repeatable reads and phantom reads are prevented.

Method Summary

void	clearWarnings() Clears all warnings reported for this <code>Connection</code> object.
void	close() Releases this <code>Connection</code> object's database and JDBC resources immediately instead of waiting for them to be automatically released.
void	commit() Makes all changes made since the previous commit/rollback permanent and releases any database locks currently held by this <code>Connection</code> object.
Statement	createStatement() Creates a <code>Statement</code> object for sending SQL statements to the database.

Statement	createStatement (int resultSetType, int resultSetConcurrency) Creates a <code>Statement</code> object that will generate <code>ResultSet</code> objects with the given type and concurrency.
Statement	createStatement (int resultSetType, int resultSetConcurrency, int resultSetHoldability) Creates a <code>Statement</code> object that will generate <code>ResultSet</code> objects with the given type, concurrency, and holdability.
boolean	getAutoCommit () Retrieves the current auto-commit mode for this <code>Connection</code> object.
String	getCatalog () Retrieves this <code>Connection</code> object's current catalog name.
int	getHoldability () Retrieves the current holdability of <code>ResultSet</code> objects created using this <code>Connection</code> object.
DatabaseMetaData	getMetaData () Retrieves a <code>DatabaseMetaData</code> object that contains metadata about the database to which this <code>Connection</code> object represents a connection.
int	getTransactionIsolation () Retrieves this <code>Connection</code> object's current transaction isolation level.
Map	getTypeMap () Retrieves the <code>Map</code> object associated with this <code>Connection</code> object.
SQLWarning	getWarnings () Retrieves the first warning reported by calls on this <code>Connection</code> object.
boolean	isClosed () Retrieves whether this <code>Connection</code> object has been closed.
boolean	isReadOnly () Retrieves whether this <code>Connection</code> object is in read-only mode.
String	nativeSQL (String sql) Converts the given SQL statement into the system's native SQL grammar.

CallableStatement	prepareCall (String sql) Creates a <code>CallableStatement</code> object for calling database stored procedures.
CallableStatement	prepareCall (String sql, int resultSetType, int resultSetConcurrency) Creates a <code>CallableStatement</code> object that will generate <code>ResultSet</code> objects with the given type and concurrency.
CallableStatement	prepareCall (String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability) Creates a <code>CallableStatement</code> object that will generate <code>ResultSet</code> objects with the given type and concurrency.
PreparedStatement	prepareStatement (String sql) Creates a <code>PreparedStatement</code> object for sending parameterized SQL statements to the database.
PreparedStatement	prepareStatement (String sql, int autoGeneratedKeys) Creates a default <code>PreparedStatement</code> object that has the capability to retrieve auto-generated keys.
PreparedStatement	prepareStatement (String sql, int[] columnIndexes) Creates a default <code>PreparedStatement</code> object capable of returning the auto-generated keys designated by the given array.
PreparedStatement	prepareStatement (String sql, int resultSetType, int resultSetConcurrency) Creates a <code>PreparedStatement</code> object that will generate <code>ResultSet</code> objects with the given type and concurrency.
PreparedStatement	prepareStatement (String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability) Creates a <code>PreparedStatement</code> object that will generate <code>ResultSet</code> objects with the given type, concurrency, and holdability.
PreparedStatement	prepareStatement (String sql, String [] columnNames) Creates a default <code>PreparedStatement</code> object capable of returning the auto-generated keys designated by the given array.
void	releaseSavepoint (Savepoint savepoint) Removes the given <code>Savepoint</code> object from the current transaction.

void	rollback() Undoes all changes made in the current transaction and releases any database locks currently held by this <code>Connection</code> object.
void	rollback(Savepoint savepoint) Undoes all changes made after the given <code>Savepoint</code> object was set.
void	setAutoCommit(boolean autoCommit) Sets this connection's auto-commit mode to the given state.
void	setCatalog(String catalog) Sets the given catalog name in order to select a subspace of this <code>Connection</code> object's database in which to work.
void	setHoldability(int holdability) Changes the holdability of <code>ResultSet</code> objects created using this <code>Connection</code> object to the given holdability.
void	setReadOnly(boolean readOnly) Puts this connection in read-only mode as a hint to the driver to enable database optimizations.
Savepoint	setSavepoint() Creates an unnamed savepoint in the current transaction and returns the new <code>Savepoint</code> object that represents it.
Savepoint	setSavepoint(String name) Creates a savepoint with the given name in the current transaction and returns the new <code>Savepoint</code> object that represents it.
void	setTransactionIsolation(int level) Attempts to change the transaction isolation level for this <code>Connection</code> object to the one given.
void	setTypeMap(Map map) Installs the given <code>TypeMap</code> object as the type map for this <code>Connection</code> object.

Field Detail

TRANSACTION_NONE

```
public static final int TRANSACTION_NONE
```

A constant indicating that transactions are not supported.

See Also:[Constant Field Values](#)

TRANSACTION_READ_UNCOMMITTED

```
public static final int TRANSACTION_READ_UNCOMMITTED
```

A constant indicating that dirty reads, non-repeatable reads and phantom reads can occur. This level allows a row changed by one transaction to be read by another transaction before any changes in that row have been committed (a "dirty read"). If any of the changes are rolled back, the second transaction will have retrieved an invalid row.

See Also:[Constant Field Values](#)

TRANSACTION_READ_COMMITTED

```
public static final int TRANSACTION_READ_COMMITTED
```

A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur. This level only prohibits a transaction from reading a row with uncommitted changes in it.

See Also:[Constant Field Values](#)

TRANSACTION_REPEATABLE_READ

```
public static final int TRANSACTION_REPEATABLE_READ
```

A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur. This level prohibits a transaction from reading a row with uncommitted changes in it, and it also prohibits the situation where one transaction reads a row, a second transaction alters the row, and the first transaction rereads the row, getting different values the second time (a "non-repeatable read").

See Also:

[Constant Field Values](#)

TRANSACTION_SERIALIZABLE

public static final int **TRANSACTION_SERIALIZABLE**

A constant indicating that dirty reads, non-repeatable reads and phantom reads are prevented. This level includes the prohibitions in `TRANSACTION_REPEATABLE_READ` and further prohibits the situation where one transaction reads all rows that satisfy a `WHERE` condition, a second transaction inserts a row that satisfies that `WHERE` condition, and the first transaction rereads for the same condition, retrieving the additional "phantom" row in the second read.

See Also:

[Constant Field Values](#)

Method Detail

createStatement

public [Statement](#) **createStatement()**
throws [SQLException](#)

Creates a `Statement` object for sending SQL statements to the database. SQL statements without parameters are normally executed using `Statement` objects. If the same SQL statement is executed many times, it may be more efficient to use a `PreparedStatement` object.

Result sets created using the returned `Statement` object will by default be type `TYPE_FORWARD_ONLY` and have a concurrency level of `CONCUR_READ_ONLY`.

Returns:

a new default `Statement` object

Throws:

[SQLException](#) - if a database access error occurs

prepareStatement

public [PreparedStatement](#) **prepareStatement([String](#) sql)**

throws [SQLException](#)

Creates a `PreparedStatement` object for sending parameterized SQL statements to the database.

A SQL statement with or without IN parameters can be pre-compiled and stored in a `PreparedStatement` object. This object can then be used to efficiently execute this statement multiple times.

Note: This method is optimized for handling parametric SQL statements that benefit from precompilation. If the driver supports precompilation, the method `prepareStatement` will send the statement to the database for precompilation. Some drivers may not support precompilation. In this case, the statement may not be sent to the database until the `PreparedStatement` object is executed. This has no direct effect on users; however, it does affect which methods throw certain `SQLException` objects.

Result sets created using the returned `PreparedStatement` object will by default be type `TYPE_FORWARD_ONLY` and have a concurrency level of `CONCUR_READ_ONLY`.

Parameters:

`sql` - an SQL statement that may contain one or more '?' IN parameter placeholders

Returns:

a new default `PreparedStatement` object containing the pre-compiled SQL statement

Throws:

[SQLException](#) - if a database access error occurs

prepareCall

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

Creates a `CallableStatement` object for calling database stored procedures. The `CallableStatement` object provides methods for setting up its IN and OUT parameters, and methods for executing the call to a stored procedure.

Note: This method is optimized for handling stored procedure call statements. Some drivers may send the call statement to the database when the method `prepareCall` is done; others may wait until the `CallableStatement` object is executed. This has no direct effect on users; however, it does affect which method throws certain `SQLExceptions`.

Result sets created using the returned `CallableStatement` object will by default be type

TYPE_FORWARD_ONLY and have a concurrency level of CONCUR_READ_ONLY.

Parameters:

`sql` - an SQL statement that may contain one or more '?' parameter placeholders. Typically this statement is a JDBC function call escape string.

Returns:

a new default `CallableStatement` object containing the pre-compiled SQL statement

Throws:

[SQLException](#) - if a database access error occurs

nativeSQL

```
public String nativeSQL(String sql)
    throws SQLException
```

Converts the given SQL statement into the system's native SQL grammar. A driver may convert the JDBC SQL grammar into its system's native SQL grammar prior to sending it. This method returns the native form of the statement that the driver would have sent.

Parameters:

`sql` - an SQL statement that may contain one or more '?' parameter placeholders

Returns:

the native form of this statement

Throws:

[SQLException](#) - if a database access error occurs

setAutoCommit

```
public void setAutoCommit(boolean autoCommit)
    throws SQLException
```

Sets this connection's auto-commit mode to the given state. If a connection is in auto-commit mode, then all its SQL statements will be executed and committed as individual transactions. Otherwise, its SQL statements are grouped into transactions that are terminated by a call to either the method `commit` or the method `rollback`. By default, new connections are in auto-commit mode.

The commit occurs when the statement completes or the next `execute` occurs, whichever comes first. In the case of statements returning a `ResultSet` object, the statement completes

when the last row of the `ResultSet` object has been retrieved or the `ResultSet` object has been closed. In advanced cases, a single statement may return multiple results as well as output parameter values. In these cases, the commit occurs when all results and output parameter values have been retrieved.

NOTE: If this method is called during a transaction, the transaction is committed.

Parameters:

`autoCommit` - `true` to enable auto-commit mode; `false` to disable it

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[getAutoCommit\(\)](#)

getAutoCommit

```
public boolean getAutoCommit()  
           throws SQLException
```

Retrieves the current auto-commit mode for this `Connection` object.

Returns:

the current state of this `Connection` object's auto-commit mode

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[setAutoCommit\(boolean\)](#)

commit

```
public void commit()  
           throws SQLException
```

Makes all changes made since the previous commit/rollback permanent and releases any database locks currently held by this `Connection` object. This method should be used only when auto-commit mode has been disabled.

Throws:

[SQLException](#) - if a database access error occurs or this `Connection` object is in

auto-commit mode

See Also:

[setAutoCommit\(boolean\)](#)

rollback

```
public void rollback()  
    throws SQLException
```

Undoes all changes made in the current transaction and releases any database locks currently held by this `Connection` object. This method should be used only when auto-commit mode has been disabled.

Throws:

[SQLException](#) - if a database access error occurs or this `Connection` object is in auto-commit mode

See Also:

[setAutoCommit\(boolean\)](#)

close

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

Releases this `Connection` object's database and JDBC resources immediately instead of waiting for them to be automatically released.

Calling the method `close` on a `Connection` object that is already closed is a no-op.

Note: A `Connection` object is automatically closed when it is garbage collected. Certain fatal errors also close a `Connection` object.

Throws:

[SQLException](#) - if a database access error occurs

isClosed

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

Retrieves whether this `Connection` object has been closed. A connection is closed if the method `close` has been called on it or if certain fatal errors have occurred. This method is guaranteed to return `true` only when it is called after the method `Connection.close` has been called.

This method generally cannot be called to determine whether a connection to a database is valid or invalid. A typical client can determine that a connection is invalid by catching any exceptions that might be thrown when an operation is attempted.

Returns:

`true` if this `Connection` object is closed; `false` if it is still open

Throws:

[SQLException](#) - if a database access error occurs

getMetaData

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

Retrieves a `DatabaseMetaData` object that contains metadata about the database to which this `Connection` object represents a connection. The metadata includes information about the database's tables, its supported SQL grammar, its stored procedures, the capabilities of this connection, and so on.

Returns:

a `DatabaseMetaData` object for this `Connection` object

Throws:

[SQLException](#) - if a database access error occurs

setReadOnly

```
public void setReadOnly(boolean readOnly)  
    throws SQLException
```

Puts this connection in read-only mode as a hint to the driver to enable database optimizations.

Note: This method cannot be called during a transaction.

Parameters:

`readOnly` - `true` enables read-only mode; `false` disables it

Throws:

[SQLException](#) - if a database access error occurs or this method is called during a transaction

isReadOnly

```
public boolean isReadOnly()  
    throws SQLException
```

Retrieves whether this `Connection` object is in read-only mode.

Returns:

`true` if this `Connection` object is read-only; `false` otherwise

Throws:

[SQLException](#) - if a database access error occurs

setCatalog

```
public void setCatalog(String catalog)  
    throws SQLException
```

Sets the given catalog name in order to select a subspace of this `Connection` object's database in which to work.

If the driver does not support catalogs, it will silently ignore this request.

Parameters:

`catalog` - the name of a catalog (subspace in this `Connection` object's database) in which to work

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[getCatalog\(\)](#)

getCatalog

```
public String getCatalog()  
           throws SQLException
```

Retrieves this `Connection` object's current catalog name.

Returns:

the current catalog name or `null` if there is none

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[setCatalog\(java.lang.String\)](#)

setTransactionIsolation

```
public void setTransactionIsolation(int level)  
           throws SQLException
```

Attempts to change the transaction isolation level for this `Connection` object to the one given. The constants defined in the interface `Connection` are the possible transaction isolation levels.

Note: If this method is called during a transaction, the result is implementation-defined.

Parameters:

`level` - one of the following `Connection` constants: `Connection.TRANSACTION_READ_UNCOMMITTED`, `Connection.TRANSACTION_READ_COMMITTED`, `Connection.TRANSACTION_REPEATABLE_READ`, or `Connection.TRANSACTION_SERIALIZABLE`. (Note that `Connection.TRANSACTION_NONE` cannot be used because it specifies that transactions are not supported.)

Throws:

[SQLException](#) - if a database access error occurs or the given parameter is not one of the `Connection` constants

See Also:

[DatabaseMetaData.supportsTransactionIsolationLevel\(int\)](#),
[getTransactionIsolation\(\)](#)

getTransactionIsolation

```
public int getTransactionIsolation()
           throws SQLException
```

Retrieves this `Connection` object's current transaction isolation level.

Returns:

the current transaction isolation level, which will be one of the following constants:
`Connection.TRANSACTION_READ_UNCOMMITTED`, `Connection.TRANSACTION_READ_COMMITTED`, `Connection.TRANSACTION_REPEATABLE_READ`, `Connection.TRANSACTION_SERIALIZABLE`, or `Connection.TRANSACTION_NONE`.

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[setTransactionIsolation\(int\)](#)

getWarnings

```
public SQLWarning getWarnings()
           throws SQLException
```

Retrieves the first warning reported by calls on this `Connection` object. If there is more than one warning, subsequent warnings will be chained to the first one and can be retrieved by calling the method `SQLWarning.getNextWarning` on the warning that was retrieved previously.

This method may not be called on a closed connection; doing so will cause an `SQLException` to be thrown.

Note: Subsequent warnings will be chained to this `SQLWarning`.

Returns:

the first `SQLWarning` object or `null` if there are none

Throws:

[SQLException](#) - if a database access error occurs or this method is called on a closed connection

See Also:

[SQLWarning](#)

clearWarnings

```
public void clearWarnings()  
           throws SQLException
```

Clears all warnings reported for this `Connection` object. After a call to this method, the method `getWarnings` returns `null` until a new warning is reported for this `Connection` object.

Throws:

[SQLException](#) - if a database access error occurs

createStatement

```
public Statement createStatement(int resultSetType,  
                                   int resultSetConcurrency)  
           throws SQLException
```

Creates a `Statement` object that will generate `ResultSet` objects with the given type and concurrency. This method is the same as the `createStatement` method above, but it allows the default result set type and concurrency to be overridden.

Parameters:

`resultSetType` - a result set type; one of `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, or `ResultSet.TYPE_SCROLL_SENSITIVE`

`resultSetConcurrency` - a concurrency type; one of `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`

Returns:

a new `Statement` object that will generate `ResultSet` objects with the given type and concurrency

Throws:

[SQLException](#) - if a database access error occurs or the given parameters are not `ResultSet` constants indicating type and concurrency

Since:

1.2

prepareStatement

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

Creates a `PreparedStatement` object that will generate `ResultSet` objects with the given type and concurrency. This method is the same as the `prepareStatement` method above, but it allows the default result set type and concurrency to be overridden.

Parameters:

`sql` - a `String` object that is the SQL statement to be sent to the database; may contain one or more ? IN parameters
`resultSetType` - a result set type; one of `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, or `ResultSet.TYPE_SCROLL_SENSITIVE`
`resultSetConcurrency` - a concurrency type; one of `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`

Returns:

a new `PreparedStatement` object containing the pre-compiled SQL statement that will produce `ResultSet` objects with the given type and concurrency

Throws:

[SQLException](#) - if a database access error occurs or the given parameters are not `ResultSet` constants indicating type and concurrency

Since:

1.2

prepareCall

```
public CallableStatement prepareCall(String sql,
                                         int resultSetType,
                                         int resultSetConcurrency)
    throws SQLException
```

Creates a `CallableStatement` object that will generate `ResultSet` objects with the given type and concurrency. This method is the same as the `prepareCall` method above, but it allows the default result set type and concurrency to be overridden.

Parameters:

`sql` - a `String` object that is the SQL statement to be sent to the database; may contain one or more ? parameters
`resultSetType` - a result set type; one of `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, or `ResultSet` .

TYPE_SCROLL_SENSITIVE

resultSetConcurrency - a concurrency type; one of `ResultSet`.

CONCUR_READ_ONLY or `ResultSet.CONCUR_UPDATABLE`

Returns:

a new `CallableStatement` object containing the pre-compiled SQL statement that will produce `ResultSet` objects with the given type and concurrency

Throws:

[SQLException](#) - if a database access error occurs or the given parameters are not `ResultSet` constants indicating type and concurrency

Since:

1.2

getTypeMap

```
public Map getTypeMap()
    throws SQLException
```

Retrieves the `Map` object associated with this `Connection` object. Unless the application has added an entry, the type map returned will be empty.

Returns:

the `java.util.Map` object associated with this `Connection` object

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

See Also:

[setTypeMap\(java.util.Map\)](#)

setTypeMap

```
public void setTypeMap(Map map)
    throws SQLException
```

Installs the given `TypeMap` object as the type map for this `Connection` object. The type map will be used for the custom mapping of SQL structured types and distinct types.

Parameters:

map - the `java.util.Map` object to install as the replacement for this `Connection` object's default type map

Throws:

[SQLException](#) - if a database access error occurs or the given parameter is not a `java.util.Map` object

Since:

1.2

See Also:

[getTypeMap\(\)](#)

setHoldability

```
public void setHoldability(int holdability)  
    throws SQLException
```

Changes the holdability of `ResultSet` objects created using this `Connection` object to the given holdability.

Parameters:

`holdability` - a `ResultSet` holdability constant; one of `ResultSet.HOLD_CURSORS_OVER_COMMIT` or `ResultSet.CLOSE_CURSORS_AT_COMMIT`

Throws:

[SQLException](#) - if a database access occurs, the given parameter is not a `ResultSet` constant indicating holdability, or the given holdability is not supported

Since:

1.4

See Also:

[getHoldability\(\)](#), [ResultSet](#)

getHoldability

```
public int getHoldability()  
    throws SQLException
```

Retrieves the current holdability of `ResultSet` objects created using this `Connection` object.

Returns:

the holdability, one of `ResultSet.HOLD_CURSORS_OVER_COMMIT` or `ResultSet.CLOSE_CURSORS_AT_COMMIT`

Throws:

[SQLException](#) - if a database access occurs

Since:

1.4

See Also:

[setHoldability\(int\)](#), [ResultSet](#)

setSavepoint

```
public Savepoint setSavepoint()  
                        throws SQLException
```

Creates an unnamed savepoint in the current transaction and returns the new [Savepoint](#) object that represents it.

Returns:

the new [Savepoint](#) object

Throws:

[SQLException](#) - if a database access error occurs or this [Connection](#) object is currently in auto-commit mode

Since:

1.4

See Also:

[Savepoint](#)

setSavepoint

```
public Savepoint setSavepoint(String name)  
                        throws SQLException
```

Creates a savepoint with the given name in the current transaction and returns the new [Savepoint](#) object that represents it.

Parameters:

name - a [String](#) containing the name of the savepoint

Returns:

the new [Savepoint](#) object

Throws:

[SQLException](#) - if a database access error occurs or this [Connection](#) object is currently in auto-commit mode

Since:

1.4

See Also:[Savepoint](#)

rollback

```
public void rollback(Savepoint savepoint)
    throws SQLException
```

Undoes all changes made after the given `Savepoint` object was set.

This method should be used only when auto-commit has been disabled.

Parameters:

savepoint - the `Savepoint` object to roll back to

Throws:

[SQLException](#) - if a database access error occurs, the `Savepoint` object is no longer valid, or this `Connection` object is currently in auto-commit mode

Since:

1.4

See Also:[Savepoint](#), [rollback\(\)](#)

releaseSavepoint

```
public void releaseSavepoint(Savepoint savepoint)
    throws SQLException
```

Removes the given `Savepoint` object from the current transaction. Any reference to the savepoint after it have been removed will cause an `SQLException` to be thrown.

Parameters:

savepoint - the `Savepoint` object to be removed

Throws:

[SQLException](#) - if a database access error occurs or the given `Savepoint` object is not a valid savepoint in the current transaction

Since:

1.4

createStatement

```
public Statement createStatement(int resultSetType,  
                                   int resultSetConcurrency,  
                                   int resultSetHoldability)  
    throws SQLException
```

Creates a `Statement` object that will generate `ResultSet` objects with the given type, concurrency, and holdability. This method is the same as the `createStatement` method above, but it allows the default result set type, concurrency, and holdability to be overridden.

Parameters:

`resultSetType` - one of the following `ResultSet` constants: `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, or `ResultSet.TYPE_SCROLL_SENSITIVE`

`resultSetConcurrency` - one of the following `ResultSet` constants: `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`

`resultSetHoldability` - one of the following `ResultSet` constants: `ResultSet.HOLD_CURSORS_OVER_COMMIT` or `ResultSet.CLOSE_CURSORS_AT_COMMIT`

Returns:

a new `Statement` object that will generate `ResultSet` objects with the given type, concurrency, and holdability

Throws:

[SQLException](#) - if a database access error occurs or the given parameters are not `ResultSet` constants indicating type, concurrency, and holdability

Since:

1.4

See Also:

[ResultSet](#)

prepareStatement

```
public PreparedStatement prepareStatement(String sql,  
                                             int resultSetType,  
                                             int resultSetConcurrency,  
                                             int resultSetHoldability)  
    throws SQLException
```

Creates a `PreparedStatement` object that will generate `ResultSet` objects with the

given type, concurrency, and holdability.

This method is the same as the `prepareStatement` method above, but it allows the default result set type, concurrency, and holdability to be overridden.

Parameters:

`sql` - a `String` object that is the SQL statement to be sent to the database; may contain one or more ? IN parameters
`resultSetType` - one of the following `ResultSet` constants: `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, or `ResultSet.TYPE_SCROLL_SENSITIVE`
`resultSetConcurrency` - one of the following `ResultSet` constants: `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`
`resultSetHoldability` - one of the following `ResultSet` constants: `ResultSet.HOLD_CURSORS_OVER_COMMIT` or `ResultSet.CLOSE_CURSORS_AT_COMMIT`

Returns:

a new `PreparedStatement` object, containing the pre-compiled SQL statement, that will generate `ResultSet` objects with the given type, concurrency, and holdability

Throws:

[SQLException](#) - if a database access error occurs or the given parameters are not `ResultSet` constants indicating type, concurrency, and holdability

Since:

1.4

See Also:

[ResultSet](#)

prepareCall

```
public CallableStatement prepareCall(String sql,
                                         int resultSetType,
                                         int resultSetConcurrency,
                                         int resultSetHoldability)
    throws SQLException
```

Creates a `CallableStatement` object that will generate `ResultSet` objects with the given type and concurrency. This method is the same as the `prepareCall` method above, but it allows the default result set type, result set concurrency type and holdability to be overridden.

Parameters:

`sql` - a `String` object that is the SQL statement to be sent to the database; may

contain one or more ? parameters

`resultSetType` - one of the following `ResultSet` constants: `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, or `ResultSet.TYPE_SCROLL_SENSITIVE`

`resultSetConcurrency` - one of the following `ResultSet` constants: `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`

`resultSetHoldability` - one of the following `ResultSet` constants: `ResultSet.HOLD_CURSORS_OVER_COMMIT` or `ResultSet.CLOSE_CURSORS_AT_COMMIT`

Returns:

a new `CallableStatement` object, containing the pre-compiled SQL statement, that will generate `ResultSet` objects with the given type, concurrency, and holdability

Throws:

[SQLException](#) - if a database access error occurs or the given parameters are not `ResultSet` constants indicating type, concurrency, and holdability

Since:

1.4

See Also:

[ResultSet](#)

prepareStatement

```
public PreparedStatement prepareStatement(String sql,
                                             int autoGeneratedKeys)
    throws SQLException
```

Creates a default `PreparedStatement` object that has the capability to retrieve auto-generated keys. The given constant tells the driver whether it should make auto-generated keys available for retrieval. This parameter is ignored if the SQL statement is not an `INSERT` statement.

Note: This method is optimized for handling parametric SQL statements that benefit from precompilation. If the driver supports precompilation, the method `prepareStatement` will send the statement to the database for precompilation. Some drivers may not support precompilation. In this case, the statement may not be sent to the database until the `PreparedStatement` object is executed. This has no direct effect on users; however, it does affect which methods throw certain `SQLExceptions`.

Result sets created using the returned `PreparedStatement` object will by default be type `TYPE_FORWARD_ONLY` and have a concurrency level of `CONCUR_READ_ONLY`.

Parameters:

`sql` - an SQL statement that may contain one or more '?' IN parameter placeholders
`autoGeneratedKeys` - a flag indicating whether auto-generated keys should be returned; one of `Statement.RETURN_GENERATED_KEYS` or `Statement.NO_GENERATED_KEYS`

Returns:

a new `PreparedStatement` object, containing the pre-compiled SQL statement, that will have the capability of returning auto-generated keys

Throws:

[SQLException](#) - if a database access error occurs or the given parameter is not a `Statement` constant indicating whether auto-generated keys should be returned

Since:

1.4

prepareStatement

```
public PreparedStatement prepareStatement(String sql,
                                             int[] columnIndexes)
    throws SQLException
```

Creates a default `PreparedStatement` object capable of returning the auto-generated keys designated by the given array. This array contains the indexes of the columns in the target table that contain the auto-generated keys that should be made available. This array is ignored if the SQL statement is not an `INSERT` statement.

An SQL statement with or without IN parameters can be pre-compiled and stored in a `PreparedStatement` object. This object can then be used to efficiently execute this statement multiple times.

Note: This method is optimized for handling parametric SQL statements that benefit from precompilation. If the driver supports precompilation, the method `prepareStatement` will send the statement to the database for precompilation. Some drivers may not support precompilation. In this case, the statement may not be sent to the database until the `PreparedStatement` object is executed. This has no direct effect on users; however, it does affect which methods throw certain `SQLExceptions`.

Result sets created using the returned `PreparedStatement` object will by default be type `TYPE_FORWARD_ONLY` and have a concurrency level of `CONCUR_READ_ONLY`.

Parameters:

`sql` - an SQL statement that may contain one or more '?' IN parameter placeholders
`columnIndexes` - an array of column indexes indicating the columns that should be returned from the inserted row or rows

Returns:

a new `PreparedStatement` object, containing the pre-compiled statement, that is capable of returning the auto-generated keys designated by the given array of column indexes

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.4

prepareStatement

```
public PreparedStatement prepareStatement(String sql,  
                                             String[] columnNames)  
    throws SQLException
```

Creates a default `PreparedStatement` object capable of returning the auto-generated keys designated by the given array. This array contains the names of the columns in the target table that contain the auto-generated keys that should be returned. This array is ignored if the SQL statement is not an `INSERT` statement.

An SQL statement with or without IN parameters can be pre-compiled and stored in a `PreparedStatement` object. This object can then be used to efficiently execute this statement multiple times.

Note: This method is optimized for handling parametric SQL statements that benefit from precompilation. If the driver supports precompilation, the method `prepareStatement` will send the statement to the database for precompilation. Some drivers may not support precompilation. In this case, the statement may not be sent to the database until the `PreparedStatement` object is executed. This has no direct effect on users; however, it does affect which methods throw certain `SQLExceptions`.

Result sets created using the returned `PreparedStatement` object will by default be type `TYPE_FORWARD_ONLY` and have a concurrency level of `CONCUR_READ_ONLY`.

Parameters:

`sql` - an SQL statement that may contain one or more '?' IN parameter placeholders
`columnNames` - an array of column names indicating the columns that should be returned from the inserted row or rows

Returns:

a new `PreparedStatement` object, containing the pre-compiled statement, that is capable of returning the auto-generated keys designated by the given array of column names

Throws:[SQLException](#) - if a database access error occurs**Since:**

1.4

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)*JavaTM 2 Platform
Std. Ed. v1.4.2*[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#)[NO FRAMES](#)[All Classes](#)SUMMARY: NESTED | [FIELD](#) | CONSTR | [METHOD](#) DETAIL: [FIELD](#) | CONSTR | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

java.sql

Interface Statement

All Known Subinterfaces:

[CallableStatement](#), [PreparedStatement](#)public interface **Statement**

The object used for executing a static SQL statement and returning the results it produces.

By default, only one `ResultSet` object per `Statement` object can be open at the same time. Therefore, if the reading of one `ResultSet` object is interleaved with the reading of another, each must have been generated by different `Statement` objects. All execution methods in the `Statement` interface implicitly close a statment's current `ResultSet` object if an open one exists.

See Also:

[Connection.createStatement\(\)](#), [ResultSet](#)

Field Summary

static int	CLOSE_ALL_RESULTS The constant indicating that all <code>ResultSet</code> objects that have previously been kept open should be closed when calling <code>getMoreResults</code> .
static int	CLOSE_CURRENT_RESULT The constant indicating that the current <code>ResultSet</code> object should be closed when calling <code>getMoreResults</code> .
static int	EXECUTE_FAILED The constant indicating that an error occured while executing a batch statement.
static int	KEEP_CURRENT_RESULT The constant indicating that the current <code>ResultSet</code> object should not be closed when calling <code>getMoreResults</code> .

static int	NO_GENERATED_KEYS The constant indicating that generated keys should not be made available for retrieval.
static int	RETURN_GENERATED_KEYS The constant indicating that generated keys should be made available for retrieval.
static int	SUCCESS_NO_INFO The constant indicating that a batch statement executed successfully but that no count of the number of rows it affected is available.

Method Summary

void	addBatch (String sql) Adds the given SQL command to the current list of commands for this Statement object.
void	cancel () Cancels this Statement object if both the DBMS and driver support aborting an SQL statement.
void	clearBatch () Empties this Statement object's current list of SQL commands.
void	clearWarnings () Clears all the warnings reported on this Statement object.
void	close () Releases this Statement object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.
boolean	execute (String sql) Executes the given SQL statement, which may return multiple results.
boolean	execute (String sql, int autoGeneratedKeys) Executes the given SQL statement, which may return multiple results, and signals the driver that any auto-generated keys should be made available for retrieval.
boolean	execute (String sql, int[] columnIndexes) Executes the given SQL statement, which may return multiple results, and signals the driver that the auto-generated keys indicated in the given array should be made available for retrieval.
boolean	execute (String sql, String [] columnNames) Executes the given SQL statement, which may return multiple results, and signals the driver that the auto-generated keys indicated in the given array should be made available for retrieval.

<code>int[]</code>	executeBatch() Submits a batch of commands to the database for execution and if all commands execute successfully, returns an array of update counts.
ResultSet	executeQuery(String sql) Executes the given SQL statement, which returns a single <code>ResultSet</code> object.
<code>int</code>	executeUpdate(String sql) Executes the given SQL statement, which may be an INSERT, UPDATE, or DELETE statement or an SQL statement that returns nothing, such as an SQL DDL statement.
<code>int</code>	executeUpdate(String sql, int autoGeneratedKeys) Executes the given SQL statement and signals the driver with the given flag about whether the auto-generated keys produced by this <code>Statement</code> object should be made available for retrieval.
<code>int</code>	executeUpdate(String sql, int[] columnIndexes) Executes the given SQL statement and signals the driver that the auto-generated keys indicated in the given array should be made available for retrieval.
<code>int</code>	executeUpdate(String sql, String[] columnNames) Executes the given SQL statement and signals the driver that the auto-generated keys indicated in the given array should be made available for retrieval.
Connection	getConnection() Retrieves the <code>Connection</code> object that produced this <code>Statement</code> object.
<code>int</code>	getFetchDirection() Retrieves the direction for fetching rows from database tables that is the default for result sets generated from this <code>Statement</code> object.
<code>int</code>	getFetchSize() Retrieves the number of result set rows that is the default fetch size for <code>ResultSet</code> objects generated from this <code>Statement</code> object.
ResultSet	getGeneratedKeys() Retrieves any auto-generated keys created as a result of executing this <code>Statement</code> object.
<code>int</code>	getMaxFieldSize() Retrieves the maximum number of bytes that can be returned for character and binary column values in a <code>ResultSet</code> object produced by this <code>Statement</code> object.
<code>int</code>	getMaxRows() Retrieves the maximum number of rows that a <code>ResultSet</code> object produced by this <code>Statement</code> object can contain.

boolean	getMoreResults() Moves to this <code>Statement</code> object's next result, returns <code>true</code> if it is a <code>ResultSet</code> object, and implicitly closes any current <code>ResultSet</code> object(s) obtained with the method <code>getResultSet</code> .
boolean	getMoreResults(int current) Moves to this <code>Statement</code> object's next result, deals with any current <code>ResultSet</code> object(s) according to the instructions specified by the given flag, and returns <code>true</code> if the next result is a <code>ResultSet</code> object.
int	getQueryTimeout() Retrieves the number of seconds the driver will wait for a <code>Statement</code> object to execute.
<code>ResultSet</code>	getResultSet() Retrieves the current result as a <code>ResultSet</code> object.
int	getResultSetConcurrency() Retrieves the result set concurrency for <code>ResultSet</code> objects generated by this <code>Statement</code> object.
int	getResultSetHoldability() Retrieves the result set holdability for <code>ResultSet</code> objects generated by this <code>Statement</code> object.
int	getResultSetType() Retrieves the result set type for <code>ResultSet</code> objects generated by this <code>Statement</code> object.
int	getUpdateCount() Retrieves the current result as an update count; if the result is a <code>ResultSet</code> object or there are no more results, -1 is returned.
<code>SQLWarning</code>	getWarnings() Retrieves the first warning reported by calls on this <code>Statement</code> object.
void	setCursorName(String name) Sets the SQL cursor name to the given <code>String</code> , which will be used by subsequent <code>Statement</code> object <code>execute</code> methods.
void	setEscapeProcessing(boolean enable) Sets escape processing on or off.
void	setFetchDirection(int direction) Gives the driver a hint as to the direction in which rows will be processed in <code>ResultSet</code> objects created using this <code>Statement</code> object.
void	setFetchSize(int rows) Gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed.

void	<code>setMaxFieldSize</code> (int max) Sets the limit for the maximum number of bytes in a <code>ResultSet</code> column storing character or binary values to the given number of bytes.
void	<code>setMaxRows</code> (int max) Sets the limit for the maximum number of rows that any <code>ResultSet</code> object can contain to the given number.
void	<code>setQueryTimeout</code> (int seconds) Sets the number of seconds the driver will wait for a <code>Statement</code> object to execute to the given number of seconds.

Field Detail

CLOSE_CURRENT_RESULT

```
public static final int CLOSE_CURRENT_RESULT
```

The constant indicating that the current `ResultSet` object should be closed when calling `getMoreResults`.

Since:

1.4

See Also:

[Constant Field Values](#)

KEEP_CURRENT_RESULT

```
public static final int KEEP_CURRENT_RESULT
```

The constant indicating that the current `ResultSet` object should not be closed when calling `getMoreResults`.

Since:

1.4

See Also:

[Constant Field Values](#)

CLOSE_ALL_RESULTS

```
public static final int CLOSE_ALL_RESULTS
```

The constant indicating that all `ResultSet` objects that have previously been kept open should be closed when calling `getMoreResults`.

Since:

1.4

See Also:

[Constant Field Values](#)

SUCCESS_NO_INFO

```
public static final int SUCCESS_NO_INFO
```

The constant indicating that a batch statement executed successfully but that no count of the number of rows it affected is available.

Since:

1.4

See Also:

[Constant Field Values](#)

EXECUTE_FAILED

```
public static final int EXECUTE_FAILED
```

The constant indicating that an error occurred while executing a batch statement.

Since:

1.4

See Also:

[Constant Field Values](#)

RETURN_GENERATED_KEYS

```
public static final int RETURN_GENERATED_KEYS
```

The constant indicating that generated keys should be made available for retrieval.

Since:

1.4

See Also:

[Constant Field Values](#)

```
NO_GENERATED_KEYS
```

```
public static final int NO_GENERATED_KEYS
```

The constant indicating that generated keys should not be made available for retrieval.

Since:

1.4

See Also:

[Constant Field Values](#)

Method Detail

executeQuery

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

Executes the given SQL statement, which returns a single `ResultSet` object.

Parameters:

`sql` - an SQL statement to be sent to the database, typically a static SQL `SELECT` statement

Returns:

a `ResultSet` object that contains the data produced by the given query; never `null`

Throws:

[SQLException](#) - if a database access error occurs or the given SQL statement produces anything other than a single `ResultSet` object

executeUpdate

```
public int executeUpdate(String sql)
                        throws SQLException
```

Executes the given SQL statement, which may be an INSERT, UPDATE, or DELETE statement or an SQL statement that returns nothing, such as an SQL DDL statement.

Parameters:

sql - an SQL INSERT, UPDATE or DELETE statement or an SQL statement that returns nothing

Returns:

either the row count for INSERT, UPDATE or DELETE statements, or 0 for SQL statements that return nothing

Throws:

[SQLException](#) - if a database access error occurs or the given SQL statement produces a `ResultSet` object

close

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

Releases this `Statement` object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed. It is generally good practice to release resources as soon as you are finished with them to avoid tying up database resources.

Calling the method `close` on a `Statement` object that is already closed has no effect.

Note: A `Statement` object is automatically closed when it is garbage collected. When a `Statement` object is closed, its current `ResultSet` object, if one exists, is also closed.

Throws:

[SQLException](#) - if a database access error occurs

getMaxFieldSize

```
public int getMaxFieldSize()
            throws SQLException
```

Retrieves the maximum number of bytes that can be returned for character and binary column values in a `ResultSet` object produced by this `Statement` object. This limit applies only to `BINARY`, `VARBINARY`, `LONGVARBINARY`, `CHAR`, `VARCHAR`, and `LONGVARCHAR` columns. If the limit is exceeded, the excess data is silently discarded.

Returns:

the current column size limit for columns storing character and binary values; zero means there is no limit

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[setMaxFieldSize\(int\)](#)

setMaxFieldSize

```
public void setMaxFieldSize(int max)
           throws SQLException
```

Sets the limit for the maximum number of bytes in a `ResultSet` column storing character or binary values to the given number of bytes. This limit applies only to `BINARY`, `VARBINARY`, `LONGVARBINARY`, `CHAR`, `VARCHAR`, and `LONGVARCHAR` fields. If the limit is exceeded, the excess data is silently discarded. For maximum portability, use values greater than 256.

Parameters:

`max` - the new column size limit in bytes; zero means there is no limit

Throws:

[SQLException](#) - if a database access error occurs or the condition `max >= 0` is not satisfied

See Also:

[getMaxFieldSize\(\)](#)

getMaxRows

```
public int getMaxRows()
           throws SQLException
```

Retrieves the maximum number of rows that a `ResultSet` object produced by this `Statement` object can contain. If this limit is exceeded, the excess rows are silently dropped.

Returns:

the current maximum number of rows for a `ResultSet` object produced by this `Statement` object; zero means there is no limit

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[setMaxRows\(int\)](#)

setMaxRows

```
public void setMaxRows(int max)
           throws SQLException
```

Sets the limit for the maximum number of rows that any `ResultSet` object can contain to the given number. If the limit is exceeded, the excess rows are silently dropped.

Parameters:

`max` - the new max rows limit; zero means there is no limit

Throws:

[SQLException](#) - if a database access error occurs or the condition `max >= 0` is not satisfied

See Also:

[getMaxRows\(\)](#)

setEscapeProcessing

```
public void setEscapeProcessing(boolean enable)
           throws SQLException
```

Sets escape processing on or off. If escape scanning is on (the default), the driver will do escape substitution before sending the SQL statement to the database. Note: Since prepared statements have usually been parsed prior to making this call, disabling escape processing for `PreparedStatement` objects will have no effect.

Parameters:

`enable` - `true` to enable escape processing; `false` to disable it

Throws:

[SQLException](#) - if a database access error occurs

getQueryTimeout

```
public int getQueryTimeout()  
        throws SQLException
```

Retrieves the number of seconds the driver will wait for a `Statement` object to execute. If the limit is exceeded, a `SQLException` is thrown.

Returns:

the current query timeout limit in seconds; zero means there is no limit

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[setQueryTimeout\(int\)](#)

setQueryTimeout

```
public void setQueryTimeout(int seconds)  
        throws SQLException
```

Sets the number of seconds the driver will wait for a `Statement` object to execute to the given number of seconds. If the limit is exceeded, an `SQLException` is thrown.

Parameters:

seconds - the new query timeout limit in seconds; zero means there is no limit

Throws:

[SQLException](#) - if a database access error occurs or the condition seconds ≥ 0 is not satisfied

See Also:

[getQueryTimeout\(\)](#)

cancel

```
public void cancel()  
        throws SQLException
```

Cancels this `Statement` object if both the DBMS and driver support aborting an SQL statement. This method can be used by one thread to cancel a statement that is being executed

by another thread.

Throws:

[SQLException](#) - if a database access error occurs

getWarnings

```
public SQLWarning getWarnings()  
                throws SQLException
```

Retrieves the first warning reported by calls on this `Statement` object. Subsequent `Statement` object warnings will be chained to this `SQLWarning` object.

The warning chain is automatically cleared each time a statement is (re)executed. This method may not be called on a closed `Statement` object; doing so will cause an `SQLException` to be thrown.

Note: If you are processing a `ResultSet` object, any warnings associated with reads on that `ResultSet` object will be chained on it rather than on the `Statement` object that produced it.

Returns:

the first `SQLWarning` object or `null` if there are no warnings

Throws:

[SQLException](#) - if a database access error occurs or this method is called on a closed statement

clearWarnings

```
public void clearWarnings()  
                throws SQLException
```

Clears all the warnings reported on this `Statement` object. After a call to this method, the method `getWarnings` will return `null` until a new warning is reported for this `Statement` object.

Throws:

[SQLException](#) - if a database access error occurs

setCursorName

```
public void setCursorName(String name)
                        throws SQLException
```

Sets the SQL cursor name to the given `String`, which will be used by subsequent `Statement` object `execute` methods. This name can then be used in SQL positioned update or delete statements to identify the current row in the `ResultSet` object generated by this statement. If the database does not support positioned update/delete, this method is a noop. To insure that a cursor has the proper isolation level to support updates, the cursor's `SELECT` statement should have the form `SELECT FOR UPDATE`. If `FOR UPDATE` is not present, positioned updates may fail.

Note: By definition, the execution of positioned updates and deletes must be done by a different `Statement` object than the one that generated the `ResultSet` object being used for positioning. Also, cursor names must be unique within a connection.

Parameters:

`name` - the new cursor name, which must be unique within a connection

Throws:

[SQLException](#) - if a database access error occurs

execute

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

Executes the given SQL statement, which may return multiple results. In some (uncommon) situations, a single SQL statement may return multiple result sets and/or update counts. Normally you can ignore this unless you are (1) executing a stored procedure that you know may return multiple results or (2) you are dynamically executing an unknown SQL string.

The `execute` method executes an SQL statement and indicates the form of the first result. You must then use the methods `getResultSet` or `getUpdateCount` to retrieve the result, and `getMoreResults` to move to any subsequent result(s).

Parameters:

`sql` - any SQL statement

Returns:

`true` if the first result is a `ResultSet` object; `false` if it is an update count or there

are no results

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[getResultSet\(\)](#), [getUpdateCount\(\)](#), [getMoreResults\(\)](#)

getResultSet

```
public ResultSet getResultSet()  
                throws SQLException
```

Retrieves the current result as a `ResultSet` object. This method should be called only once per result.

Returns:

the current result as a `ResultSet` object or `null` if the result is an update count or there are no more results

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[execute\(java.lang.String\)](#)

getUpdateCount

```
public int getUpdateCount()  
          throws SQLException
```

Retrieves the current result as an update count; if the result is a `ResultSet` object or there are no more results, -1 is returned. This method should be called only once per result.

Returns:

the current result as an update count; -1 if the current result is a `ResultSet` object or there are no more results

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[execute\(java.lang.String\)](#)

getMoreResults

```
public boolean getMoreResults()
                throws SQLException
```

Moves to this `Statement` object's next result, returns `true` if it is a `ResultSet` object, and implicitly closes any current `ResultSet` object(s) obtained with the method `getResultSet`.

There are no more results when the following is true:

```
// stmt is a Statement object
((stmt.getMoreResults() == false) && (stmt.getUpdateCount
() == -1))
```

Returns:

`true` if the next result is a `ResultSet` object; `false` if it is an update count or there are no more results

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[execute\(java.lang.String\)](#)

setFetchDirection

```
public void setFetchDirection(int direction)
                throws SQLException
```

Gives the driver a hint as to the direction in which rows will be processed in `ResultSet` objects created using this `Statement` object. The default value is `ResultSet.FETCH_FORWARD`.

Note that this method sets the default fetch direction for result sets generated by this `Statement` object. Each result set has its own methods for getting and setting its own fetch direction.

Parameters:

`direction` - the initial direction for processing rows

Throws:

[SQLException](#) - if a database access error occurs or the given direction is not one of `ResultSet.FETCH_FORWARD`, `ResultSet.FETCH_REVERSE`, or

`ResultSet.FETCH_UNKNOWN`**Since:**

1.2

See Also:[getFetchDirection\(\)](#)

getFetchDirection

```
public int getFetchDirection()  
           throws SQLException
```

Retrieves the direction for fetching rows from database tables that is the default for result sets generated from this `Statement` object. If this `Statement` object has not set a fetch direction by calling the method `setFetchDirection`, the return value is implementation-specific.

Returns:the default fetch direction for result sets generated from this `Statement` object**Throws:**[SQLException](#) - if a database access error occurs**Since:**

1.2

See Also:[setFetchDirection\(int\)](#)

setFetchSize

```
public void setFetchSize(int rows)  
           throws SQLException
```

Gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed. The number of rows specified affects only result sets created using this statement. If the value specified is zero, then the hint is ignored. The default value is zero.

Parameters:

rows - the number of rows to fetch

Throws:[SQLException](#) - if a database access error occurs, or the condition `0 <= rows <= this.getMaxRows()` is not satisfied.

Since:

1.2

See Also:[getFetchSize\(\)](#)

getFetchSize

```
public int getFetchSize()  
        throws SQLException
```

Retrieves the number of result set rows that is the default fetch size for `ResultSet` objects generated from this `Statement` object. If this `Statement` object has not set a fetch size by calling the method `setFetchSize`, the return value is implementation-specific.

Returns:

the default fetch size for result sets generated from this `Statement` object

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

See Also:[setFetchSize\(int\)](#)

getResultSetConcurrency

```
public int getResultSetConcurrency()  
        throws SQLException
```

Retrieves the result set concurrency for `ResultSet` objects generated by this `Statement` object.

Returns:

either `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getResultSetType

```
public int getResultSetType()  
        throws SQLException
```

Retrieves the result set type for `ResultSet` objects generated by this `Statement` object.

Returns:

one of `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, or `ResultSet.TYPE_SCROLL_SENSITIVE`

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

addBatch

```
public void addBatch(String sql)  
        throws SQLException
```

Adds the given SQL command to the current list of commands for this `Statement` object. The commands in this list can be executed as a batch by calling the method `executeBatch`.

NOTE: This method is optional.

Parameters:

`sql` - typically this is a static SQL `INSERT` or `UPDATE` statement

Throws:

[SQLException](#) - if a database access error occurs, or the driver does not support batch updates

Since:

1.2

See Also:

[executeBatch\(\)](#)

clearBatch

```
public void clearBatch()  
        throws SQLException
```

Empties this **Statement** object's current list of SQL commands.

NOTE: This method is optional.

Throws:

[SQLException](#) - if a database access error occurs or the driver does not support batch updates

Since:

1.2

See Also:

[addBatch\(java.lang.String\)](#)

executeBatch

```
public int[] executeBatch()
           throws SQLException
```

Submits a batch of commands to the database for execution and if all commands execute successfully, returns an array of update counts. The `int` elements of the array that is returned are ordered to correspond to the commands in the batch, which are ordered according to the order in which they were added to the batch. The elements in the array returned by the method `executeBatch` may be one of the following:

1. A number greater than or equal to zero -- indicates that the command was processed successfully and is an update count giving the number of rows in the database that were affected by the command's execution
2. A value of `SUCCESS_NO_INFO` -- indicates that the command was processed successfully but that the number of rows affected is unknown

If one of the commands in a batch update fails to execute properly, this method throws a `BatchUpdateException`, and a JDBC driver may or may not continue to process the remaining commands in the batch. However, the driver's behavior must be consistent with a particular DBMS, either always continuing to process commands or never continuing to process commands. If the driver continues processing after a failure, the array returned by the method `BatchUpdateException.getUpdateCounts` will contain as many elements as there are commands in the batch, and at least one of the elements will be the following:

3. A value of `EXECUTE_FAILED` -- indicates that the command failed to execute successfully and occurs only if a driver continues to process commands after a command fails

A driver is not required to implement this method. The possible implementations and return

values have been modified in the Java 2 SDK, Standard Edition, version 1.3 to accommodate the option of continuing to process commands in a batch update after a `BatchUpdateException` object has been thrown.

Returns:

an array of update counts containing one element for each command in the batch. The elements of the array are ordered according to the order in which commands were added to the batch.

Throws:

[SQLException](#) - if a database access error occurs or the driver does not support batch statements. Throws [BatchUpdateException](#) (a subclass of [SQLException](#)) if one of the commands sent to the database fails to execute properly or attempts to return a result set.

Since:

1.3

getConnection

```
public Connection getConnection()  
           throws SQLException
```

Retrieves the `Connection` object that produced this `Statement` object.

Returns:

the connection that produced this statement

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getMoreResults

```
public boolean getMoreResults(int current)  
           throws SQLException
```

Moves to this `Statement` object's next result, deals with any current `ResultSet` object(s) according to the instructions specified by the given flag, and returns `true` if the next result is a `ResultSet` object.

There are no more results when the following is true:

```
// stmt is a Statement object
((stmt.getMoreResults() == false) && (stmt.getUpdateCount() == -1))
```

Parameters:

`current` - one of the following `Statement` constants indicating what should happen to current `ResultSet` objects obtained using the method `getResultSet`: `Statement.CLOSE_CURRENT_RESULT`, `Statement.KEEP_CURRENT_RESULT`, or `Statement.CLOSE_ALL_RESULTS`

Returns:

`true` if the next result is a `ResultSet` object; `false` if it is an update count or there are no more results

Throws:

[SQLException](#) - if a database access error occurs or the argument supplied is not one of the following: `Statement.CLOSE_CURRENT_RESULT`, `Statement.KEEP_CURRENT_RESULT`, or `Statement.CLOSE_ALL_RESULTS`

Since:

1.4

See Also:

[execute\(java.lang.String\)](#)

getGeneratedKeys

```
public ResultSet getGeneratedKeys()
                        throws SQLException
```

Retrieves any auto-generated keys created as a result of executing this `Statement` object. If this `Statement` object did not generate any keys, an empty `ResultSet` object is returned.

Returns:

a `ResultSet` object containing the auto-generated key(s) generated by the execution of this `Statement` object

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.4

executeUpdate

```
public int executeUpdate(String sql,
                        int autoGeneratedKeys)
    throws SQLException
```

Executes the given SQL statement and signals the driver with the given flag about whether the auto-generated keys produced by this `Statement` object should be made available for retrieval.

Parameters:

`sql` - must be an SQL INSERT, UPDATE or DELETE statement or an SQL statement that returns nothing

`autoGeneratedKeys` - a flag indicating whether auto-generated keys should be made available for retrieval; one of the following constants: `Statement.RETURN_GENERATED_KEYS` `Statement.NO_GENERATED_KEYS`

Returns:

either the row count for INSERT, UPDATE or DELETE statements, or 0 for SQL statements that return nothing

Throws:

[SQLException](#) - if a database access error occurs, the given SQL statement returns a `ResultSet` object, or the given constant is not one of those allowed

Since:

1.4

executeUpdate

```
public int executeUpdate(String sql,
                        int[] columnIndexes)
    throws SQLException
```

Executes the given SQL statement and signals the driver that the auto-generated keys indicated in the given array should be made available for retrieval. The driver will ignore the array if the SQL statement is not an INSERT statement.

Parameters:

`sql` - an SQL INSERT, UPDATE or DELETE statement or an SQL statement that returns nothing, such as an SQL DDL statement

`columnIndexes` - an array of column indexes indicating the columns that should be returned from the inserted row

Returns:

either the row count for INSERT, UPDATE, or DELETE statements, or 0 for SQL statements that return nothing

Throws:

[SQLException](#) - if a database access error occurs, the SQL statement returns a `ResultSet` object, or the second argument supplied to this method is not an `int` array whose elements are valid column indexes

Since:

1.4

executeUpdate

```
public int executeUpdate(String sql,
                        String[] columnNames)
    throws SQLException
```

Executes the given SQL statement and signals the driver that the auto-generated keys indicated in the given array should be made available for retrieval. The driver will ignore the array if the SQL statement is not an `INSERT` statement.

Parameters:

`sql` - an SQL `INSERT`, `UPDATE` or `DELETE` statement or an SQL statement that returns nothing

`columnNames` - an array of the names of the columns that should be returned from the inserted row

Returns:

either the row count for `INSERT`, `UPDATE`, or `DELETE` statements, or 0 for SQL statements that return nothing

Throws:

[SQLException](#) - if a database access error occurs, the SQL statement returns a `ResultSet` object, or the second argument supplied to this method is not a `String` array whose elements are valid column names

Since:

1.4

execute

```
public boolean execute(String sql,
                     int autoGeneratedKeys)
    throws SQLException
```

Executes the given SQL statement, which may return multiple results, and signals the driver that any auto-generated keys should be made available for retrieval. The driver will ignore this signal if the SQL statement is not an `INSERT` statement.

In some (uncommon) situations, a single SQL statement may return multiple result sets and/or update counts. Normally you can ignore this unless you are (1) executing a stored procedure that you know may return multiple results or (2) you are dynamically executing an unknown SQL string.

The `execute` method executes an SQL statement and indicates the form of the first result. You must then use the methods `getResultSet` or `getUpdateCount` to retrieve the result, and `getMoreResults` to move to any subsequent result(s).

Parameters:

`sql` - any SQL statement

`autoGeneratedKeys` - a constant indicating whether auto-generated keys should be made available for retrieval using the method `getGeneratedKeys`; one of the following constants: `Statement.RETURN_GENERATED_KEYS` or `Statement.NO_GENERATED_KEYS`

Returns:

`true` if the first result is a `ResultSet` object; `false` if it is an update count or there are no results

Throws:

[SQLException](#) - if a database access error occurs or the second parameter supplied to this method is not `Statement.RETURN_GENERATED_KEYS` or `Statement.NO_GENERATED_KEYS`.

Since:

1.4

See Also:

[getResultSet\(\)](#), [getUpdateCount\(\)](#), [getMoreResults\(\)](#), [getGeneratedKeys\(\)](#)

execute

```
public boolean execute(String sql,
                      int[] columnIndexes)
    throws SQLException
```

Executes the given SQL statement, which may return multiple results, and signals the driver that the auto-generated keys indicated in the given array should be made available for retrieval. This array contains the indexes of the columns in the target table that contain the auto-generated keys that should be made available. The driver will ignore the array if the given SQL statement is not an `INSERT` statement.

Under some (uncommon) situations, a single SQL statement may return multiple result sets

and/or update counts. Normally you can ignore this unless you are (1) executing a stored procedure that you know may return multiple results or (2) you are dynamically executing an unknown SQL string.

The `execute` method executes an SQL statement and indicates the form of the first result. You must then use the methods `getResultSet` or `getUpdateCount` to retrieve the result, and `getMoreResults` to move to any subsequent result(s).

Parameters:

`sql` - any SQL statement

`columnIndexes` - an array of the indexes of the columns in the inserted row that should be made available for retrieval by a call to the method `getGeneratedKeys`

Returns:

`true` if the first result is a `ResultSet` object; `false` if it is an update count or there are no results

Throws:

[SQLException](#) - if a database access error occurs or the elements in the `int` array passed to this method are not valid column indexes

Since:

1.4

See Also:

[getResultSet\(\)](#), [getUpdateCount\(\)](#), [getMoreResults\(\)](#)

execute

```
public boolean execute(String sql,
                      String[] columnNames)
    throws SQLException
```

Executes the given SQL statement, which may return multiple results, and signals the driver that the auto-generated keys indicated in the given array should be made available for retrieval. This array contains the names of the columns in the target table that contain the auto-generated keys that should be made available. The driver will ignore the array if the given SQL statement is not an `INSERT` statement.

In some (uncommon) situations, a single SQL statement may return multiple result sets and/or update counts. Normally you can ignore this unless you are (1) executing a stored procedure that you know may return multiple results or (2) you are dynamically executing an unknown SQL string.

The `execute` method executes an SQL statement and indicates the form of the first result. You must then use the methods `getResultSet` or `getUpdateCount` to retrieve the

result, and `getMoreResults` to move to any subsequent result(s).

Parameters:

`sql` - any SQL statement

`columnNames` - an array of the names of the columns in the inserted row that should be made available for retrieval by a call to the method `getGeneratedKeys`

Returns:

`true` if the next result is a `ResultSet` object; `false` if it is an update count or there are no more results

Throws:

[SQLException](#) - if a database access error occurs or the elements of the `String` array passed to this method are not valid column names

Since:

1.4

See Also:

[getResultSet\(\)](#), [getUpdateCount\(\)](#), [getMoreResults\(\)](#),
[getGeneratedKeys\(\)](#)

getResultSetHoldability

```
public int getResultSetHoldability()
           throws SQLException
```

Retrieves the result set holdability for `ResultSet` objects generated by this `Statement` object.

Returns:

either `ResultSet.HOLD_CURSORS_OVER_COMMIT` or `ResultSet.CLOSE_CURSORS_AT_COMMIT`

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.4

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | [FIELD](#) | CONSTR | [METHOD](#) DETAIL: [FIELD](#) | CONSTR | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews,

definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

java.sql

Interface PreparedStatement

All Superinterfaces:

[Statement](#)

All Known Subinterfaces:

[CallableStatement](#)

public interface **PreparedStatement**extends [Statement](#)

An object that represents a precompiled SQL statement.

A SQL statement is precompiled and stored in a **PreparedStatement** object. This object can then be used to efficiently execute this statement multiple times.

Note: The setter methods (`setShort`, `setString`, and so on) for setting IN parameter values must specify types that are compatible with the defined SQL type of the input parameter. For instance, if the IN parameter has SQL type **INTEGER**, then the method `setInt` should be used.

If arbitrary parameter type conversions are required, the method `setObject` should be used with a target SQL type.

In the following example of setting a parameter, `con` represents an active connection:

```
PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES
                                                SET SALARY = ? WHERE ID = ?");
pstmt.setBigDecimal(1, 153833.00)
pstmt.setInt(2, 110592)
```

See Also:

[Connection.prepareStatement\(java.lang.String\)](#), [ResultSet](#)

Field Summary

Fields inherited from interface [java.sql.Statement](#)

[CLOSE ALL RESULTS](#), [CLOSE CURRENT RESULT](#), [EXECUTE FAILED](#),
[KEEP CURRENT RESULT](#), [NO GENERATED KEYS](#), [RETURN GENERATED KEYS](#),
[SUCCESS NO INFO](#)

Method Summary

void	addBatch() Adds a set of parameters to this PreparedStatement object's batch of commands.
void	clearParameters() Clears the current parameter values immediately.
boolean	execute() Executes the SQL statement in this PreparedStatement object, which may be any kind of SQL statement.
ResultSet	executeQuery() Executes the SQL query in this PreparedStatement object and returns the ResultSet object generated by the query.
int	executeUpdate() Executes the SQL statement in this PreparedStatement object, which must be an SQL INSERT, UPDATE or DELETE statement; or an SQL statement that returns nothing, such as a DDL statement.
ResultSetMetaData	getMetaData() Retrieves a ResultSetMetaData object that contains information about the columns of the ResultSet object that will be returned when this PreparedStatement object is executed.
ParameterMetaData	getParameterMetaData() Retrieves the number, types and properties of this PreparedStatement object's parameters.
void	setArray(int i, Array x) Sets the designated parameter to the given Array object.
void	setAsciiStream(int parameterIndex, InputStream x, int length) Sets the designated parameter to the given input stream, which will have the specified number of bytes.

void	setBigDecimal (int parameterIndex, BigDecimal x) Sets the designated parameter to the given java.math.BigDecimal value.
void	setBinaryStream (int parameterIndex, InputStream x, int length) Sets the designated parameter to the given input stream, which will have the specified number of bytes.
void	setBlob (int i, Blob x) Sets the designated parameter to the given Blob object.
void	setBoolean (int parameterIndex, boolean x) Sets the designated parameter to the given Java boolean value.
void	setByte (int parameterIndex, byte x) Sets the designated parameter to the given Java byte value.
void	setBytes (int parameterIndex, byte[] x) Sets the designated parameter to the given Java array of bytes.
void	setCharacterStream (int parameterIndex, Reader reader, int length) Sets the designated parameter to the given Reader object, which is the given number of characters long.
void	setClob (int i, Clob x) Sets the designated parameter to the given Clob object.
void	setDate (int parameterIndex, Date x) Sets the designated parameter to the given java.sql.Date value.
void	setDate (int parameterIndex, Date x, Calendar cal) Sets the designated parameter to the given java.sql.Date value, using the given Calendar object.
void	setDouble (int parameterIndex, double x) Sets the designated parameter to the given Java double value.
void	setFloat (int parameterIndex, float x) Sets the designated parameter to the given Java float value.
void	setInt (int parameterIndex, int x) Sets the designated parameter to the given Java int value.
void	setLong (int parameterIndex, long x) Sets the designated parameter to the given Java long value.
void	setNull (int parameterIndex, int sqlType) Sets the designated parameter to SQL NULL.

void	<code>setNull</code> (int paramIndex, int sqlType, <code>String</code> typeName) Sets the designated parameter to SQL NULL.
void	<code>setObject</code> (int parameterIndex, <code>Object</code> x) Sets the value of the designated parameter using the given object.
void	<code>setObject</code> (int parameterIndex, <code>Object</code> x, int targetSqlType) Sets the value of the designated parameter with the given object.
void	<code>setObject</code> (int parameterIndex, <code>Object</code> x, int targetSqlType, int scale) Sets the value of the designated parameter with the given object.
void	<code>setRef</code> (int i, <code>Ref</code> x) Sets the designated parameter to the given REF(<structured-type>) value.
void	<code>setShort</code> (int parameterIndex, short x) Sets the designated parameter to the given Java short value.
void	<code>setString</code> (int parameterIndex, <code>String</code> x) Sets the designated parameter to the given Java String value.
void	<code>setTime</code> (int parameterIndex, <code>Time</code> x) Sets the designated parameter to the given java.sql.Time value.
void	<code>setTime</code> (int parameterIndex, <code>Time</code> x, <code>Calendar</code> cal) Sets the designated parameter to the given java.sql.Time value, using the given Calendar object.
void	<code>setTimestamp</code> (int parameterIndex, <code>Timestamp</code> x) Sets the designated parameter to the given java.sql.Timestamp value.
void	<code>setTimestamp</code> (int parameterIndex, <code>Timestamp</code> x, <code>Calendar</code> cal) Sets the designated parameter to the given java.sql.Timestamp value, using the given Calendar object.
void	<code>setUnicodeStream</code> (int parameterIndex, <code>InputStream</code> x, int length) Deprecated.
void	<code>setURL</code> (int parameterIndex, <code>URL</code> x) Sets the designated parameter to the given java.net.URL value.

Methods inherited from interface [java.sql.Statement](#)

[addBatch](#), [cancel](#), [clearBatch](#), [clearWarnings](#), [close](#), [execute](#),
[execute](#), [execute](#), [execute](#), [executeBatch](#), [executeQuery](#),
[executeUpdate](#), [executeUpdate](#), [executeUpdate](#), [executeUpdate](#),
[getConnection](#), [getFetchDirection](#), [getFetchSize](#), [getGeneratedKeys](#),
[getMaxFieldSize](#), [getMaxRows](#), [getMoreResults](#), [getMoreResults](#),
[getQueryTimeout](#), [getResultSet](#), [getResultSetConcurrency](#),
[getResultSetHoldability](#), [getResultSetType](#), [getUpdateCount](#),
[getWarnings](#), [setCursorName](#), [setEscapeProcessing](#),
[setFetchDirection](#), [setFetchSize](#), [setMaxFieldSize](#), [setMaxRows](#),
[setQueryTimeout](#)

Method Detail**executeQuery**

```
public ResultSet executeQuery()
        throws SQLException
```

Executes the SQL query in this `PreparedStatement` object and returns the `ResultSet` object generated by the query.

Returns:

a `ResultSet` object that contains the data produced by the query; never null

Throws:

[SQLException](#) - if a database access error occurs or the SQL statement does not return a `ResultSet` object

executeUpdate

```
public int executeUpdate()
        throws SQLException
```

Executes the SQL statement in this `PreparedStatement` object, which must be an SQL INSERT, UPDATE or DELETE statement; or an SQL statement that returns nothing, such as a DDL statement.

Returns:

either (1) the row count for INSERT, UPDATE, or DELETE statements or (2) 0 for SQL statements that return nothing

Throws:

[SQLException](#) - if a database access error occurs or the SQL statement returns a `ResultSet` object

setNull

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

Sets the designated parameter to SQL NULL.

Note: You must specify the parameter's SQL type.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...
`sqlType` - the SQL type code defined in `java.sql.Types`

Throws:

[SQLException](#) - if a database access error occurs

setBoolean

```
public void setBoolean(int parameterIndex,  
                       boolean x)  
    throws SQLException
```

Sets the designated parameter to the given Java `boolean` value. The driver converts this to an SQL BIT value when it sends it to the database.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...
`x` - the parameter value

Throws:

[SQLException](#) - if a database access error occurs

setByte

```
public void setByte(int parameterIndex,  
                    byte x)  
    throws SQLException
```

Sets the designated parameter to the given Java `byte` value. The driver converts this to an SQL `TINYINT` value when it sends it to the database.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...
`x` - the parameter value

Throws:

[SQLException](#) - if a database access error occurs

setShort

```
public void setShort(int parameterIndex,  
                     short x)  
    throws SQLException
```

Sets the designated parameter to the given Java `short` value. The driver converts this to an SQL `SMALLINT` value when it sends it to the database.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...
`x` - the parameter value

Throws:

[SQLException](#) - if a database access error occurs

setInt

```
public void setInt(int parameterIndex,  
                   int x)  
    throws SQLException
```

Sets the designated parameter to the given Java `int` value. The driver converts this to an SQL `INTEGER` value when it sends it to the database.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the parameter value

Throws:

[SQLException](#) - if a database access error occurs

setLong

```
public void setLong(int parameterIndex,  
                    long x)  
    throws SQLException
```

Sets the designated parameter to the given Java `long` value. The driver converts this to an SQL BIGINT value when it sends it to the database.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the parameter value

Throws:

[SQLException](#) - if a database access error occurs

setFloat

```
public void setFloat(int parameterIndex,  
                    float x)  
    throws SQLException
```

Sets the designated parameter to the given Java `float` value. The driver converts this to an SQL FLOAT value when it sends it to the database.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the parameter value

Throws:

[SQLException](#) - if a database access error occurs

setDouble


```
public void setDouble(int parameterIndex,  
                      double x)  
    throws SQLException
```

Sets the designated parameter to the given Java `double` value. The driver converts this to an SQL `DOUBLE` value when it sends it to the database.

Parameters:

parameterIndex - the first parameter is 1, the second is 2, ...
x - the parameter value

Throws:

[SQLException](#) - if a database access error occurs

setBigDecimal

```
public void setBigDecimal(int parameterIndex,  
                           BigDecimal x)  
    throws SQLException
```

Sets the designated parameter to the given `java.math.BigDecimal` value. The driver converts this to an SQL `NUMERIC` value when it sends it to the database.

Parameters:

parameterIndex - the first parameter is 1, the second is 2, ...
x - the parameter value

Throws:

[SQLException](#) - if a database access error occurs

setString

```
public void setString(int parameterIndex,  
                      String x)  
    throws SQLException
```

Sets the designated parameter to the given Java `String` value. The driver converts this to an SQL `VARCHAR` or `LONGVARCHAR` value (depending on the argument's size relative to the driver's limits on `VARCHAR` values) when it sends it to the database.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the parameter value

Throws:

[SQLException](#) - if a database access error occurs

setBytes

```
public void setBytes(int parameterIndex,  
                    byte[] x)  
    throws SQLException
```

Sets the designated parameter to the given Java array of bytes. The driver converts this to an SQL VARBINARY or LONGVARBINARY (depending on the argument's size relative to the driver's limits on VARBINARY values) when it sends it to the database.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the parameter value

Throws:

[SQLException](#) - if a database access error occurs

setDate

```
public void setDate(int parameterIndex,  
                   Date x)  
    throws SQLException
```

Sets the designated parameter to the given `java.sql.Date` value. The driver converts this to an SQL DATE value when it sends it to the database.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the parameter value

Throws:

[SQLException](#) - if a database access error occurs

setTime

```
public void setTime(int parameterIndex,  
                   Time x)  
    throws SQLException
```

Sets the designated parameter to the given `java.sql.Time` value. The driver converts this to an SQL TIME value when it sends it to the database.

Parameters:

parameterIndex - the first parameter is 1, the second is 2, ...

x - the parameter value

Throws:

[SQLException](#) - if a database access error occurs

setTimestamp

```
public void setTimestamp(int parameterIndex,  
                        Timestamp x)  
    throws SQLException
```

Sets the designated parameter to the given `java.sql.Timestamp` value. The driver converts this to an SQL TIMESTAMP value when it sends it to the database.

Parameters:

parameterIndex - the first parameter is 1, the second is 2, ...

x - the parameter value

Throws:

[SQLException](#) - if a database access error occurs

setAsciiStream

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

Sets the designated parameter to the given input stream, which will have the specified number of bytes. When a very large ASCII value is input to a LONGVARCHAR parameter, it may be more practical to send it via a `java.io.InputStream`. Data will be read from the stream as needed until end-of-file is reached. The JDBC driver will do any necessary conversion from

ASCII to the database char format.

Note: This stream object can either be a standard Java stream object or your own subclass that implements the standard interface.

Parameters:

parameterIndex - the first parameter is 1, the second is 2, ...
 x - the Java input stream that contains the ASCII parameter value
 length - the number of bytes in the stream

Throws:

[SQLException](#) - if a database access error occurs

setUnicodeStream

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

Deprecated.

Sets the designated parameter to the given input stream, which will have the specified number of bytes. A Unicode character has two bytes, with the first byte being the high byte, and the second being the low byte. When a very large Unicode value is input to a LONGVARCHAR parameter, it may be more practical to send it via a `java.io.InputStream` object. The data will be read from the stream as needed until end-of-file is reached. The JDBC driver will do any necessary conversion from Unicode to the database char format.

Note: This stream object can either be a standard Java stream object or your own subclass that implements the standard interface.

Parameters:

parameterIndex - the first parameter is 1, the second is 2, ...
 x - a `java.io.InputStream` object that contains the Unicode parameter value as two-byte Unicode characters
 length - the number of bytes in the stream

Throws:

[SQLException](#) - if a database access error occurs

setBinaryStream

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

Sets the designated parameter to the given input stream, which will have the specified number of bytes. When a very large binary value is input to a `LONGVARBINARY` parameter, it may be more practical to send it via a `java.io.InputStream` object. The data will be read from the stream as needed until end-of-file is reached.

Note: This stream object can either be a standard Java stream object or your own subclass that implements the standard interface.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the java input stream which contains the binary parameter value

`length` - the number of bytes in the stream

Throws:

[SQLException](#) - if a database access error occurs

clearParameters

```
public void clearParameters()
    throws SQLException
```

Clears the current parameter values immediately.

In general, parameter values remain in force for repeated use of a statement. Setting a parameter value automatically clears its previous value. However, in some cases it is useful to immediately release the resources used by the current parameter values; this can be done by calling the method `clearParameters`.

Throws:

[SQLException](#) - if a database access error occurs

setObject

```
public void setObject(int parameterIndex,
                       Object x,
```

```

        int targetType,
        int scale)
    throws SQLException

```

Sets the value of the designated parameter with the given object. The second argument must be an object type; for integral values, the `java.lang` equivalent objects should be used.

The given Java object will be converted to the given `targetSqlType` before being sent to the database. If the object has a custom mapping (is of a class implementing the interface `SQLData`), the JDBC driver should call the method `SQLData.writeSQL` to write it to the SQL data stream. If, on the other hand, the object is of a class implementing `Ref`, `Blob`, `Clob`, `Struct`, or `Array`, the driver should pass it to the database as a value of the corresponding SQL type.

Note that this method may be used to pass database-specific abstract data types.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the object containing the input parameter value

`targetSqlType` - the SQL type (as defined in `java.sql.Types`) to be sent to the database. The scale argument may further qualify this type.

`scale` - for `java.sql.Types.DECIMAL` or `java.sql.Types.NUMERIC` types, this is the number of digits after the decimal point. For all other types, this value will be ignored.

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[Types](#)

setObject

```

public void setObject(int parameterIndex,
                    Object x,
                    int targetType)
    throws SQLException

```

Sets the value of the designated parameter with the given object. This method is like the method `setObject` above, except that it assumes a scale of zero.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the object containing the input parameter value

`targetSqlType` - the SQL type (as defined in `java.sql.Types`) to be sent to the

database

Throws:

[SQLException](#) - if a database access error occurs

setObject

```
public void setObject(int parameterIndex,  
                      Object x)  
    throws SQLException
```

Sets the value of the designated parameter using the given object. The second parameter must be of type `Object`; therefore, the `java.lang` equivalent objects should be used for built-in types.

The JDBC specification specifies a standard mapping from Java `Object` types to SQL types. The given argument will be converted to the corresponding SQL type before being sent to the database.

Note that this method may be used to pass database- specific abstract data types, by using a driver-specific Java type. If the object is of a class implementing the interface `SQLData`, the JDBC driver should call the method `SQLData.writeSQL` to write it to the SQL data stream. If, on the other hand, the object is of a class implementing `Ref`, `Blob`, `Clob`, `Struct`, or `Array`, the driver should pass it to the database as a value of the corresponding SQL type.

This method throws an exception if there is an ambiguity, for example, if the object is of a class implementing more than one of the interfaces named above.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the object containing the input parameter value

Throws:

[SQLException](#) - if a database access error occurs or the type of the given object is ambiguous

execute

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

Executes the SQL statement in this `PreparedStatement` object, which may be any kind of SQL statement. Some prepared statements return multiple results; the `execute` method handles these complex statements as well as the simpler form of statements handled by the methods `executeQuery` and `executeUpdate`.

The `execute` method returns a `boolean` to indicate the form of the first result. You must call either the method `getResultSet` or `getUpdateCount` to retrieve the result; you must call `getMoreResults` to move to any subsequent result(s).

Returns:

`true` if the first result is a `ResultSet` object; `false` if the first result is an update count or there is no result

Throws:

[SQLException](#) - if a database access error occurs or an argument is supplied to this method

See Also:

[Statement.execute\(java.lang.String\)](#), [Statement.getResultSet\(\)](#), [Statement.getUpdateCount\(\)](#), [Statement.getMoreResults\(\)](#)

addBatch

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

Adds a set of parameters to this `PreparedStatement` object's batch of commands.

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

See Also:

[Statement.addBatch\(java.lang.String\)](#)

setCharacterStream

```
public void setCharacterStream(int parameterIndex,  
                               Reader reader,  
                               int length)  
           throws SQLException
```


Sets the designated parameter to the given `Reader` object, which is the given number of characters long. When a very large UNICODE value is input to a `LONGVARCHAR` parameter, it may be more practical to send it via a `java.io.Reader` object. The data will be read from the stream as needed until end-of-file is reached. The JDBC driver will do any necessary conversion from UNICODE to the database char format.

Note: This stream object can either be a standard Java stream object or your own subclass that implements the standard interface.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...
`reader` - the `java.io.Reader` object that contains the Unicode data
`length` - the number of characters in the stream

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

setRef

```
public void setRef(int i,
                  Ref x)
    throws SQLException
```

Sets the designated parameter to the given `REF(<structured-type>)` value. The driver converts this to an SQL REF value when it sends it to the database.

Parameters:

`i` - the first parameter is 1, the second is 2, ...
`x` - an SQL REF value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

setBlob

```
public void setBlob(int i,
                   Blob x)
    throws SQLException
```

Sets the designated parameter to the given `Blob` object. The driver converts this to an SQL BLOB value when it sends it to the database.

Parameters:

`i` - the first parameter is 1, the second is 2, ...

`x` - a `Blob` object that maps an SQL BLOB value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

setClob

```
public void setClob(int i,  
                    Clob x)  
    throws SQLException
```

Sets the designated parameter to the given `Clob` object. The driver converts this to an SQL CLOB value when it sends it to the database.

Parameters:

`i` - the first parameter is 1, the second is 2, ...

`x` - a `Clob` object that maps an SQL CLOB value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

setArray

```
public void setArray(int i,  
                     Array x)  
    throws SQLException
```

Sets the designated parameter to the given `Array` object. The driver converts this to an SQL ARRAY value when it sends it to the database.

Parameters:

`i` - the first parameter is 1, the second is 2, ...

`x` - an `Array` object that maps an SQL `ARRAY` value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getMetaData

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

Retrieves a `ResultSetMetaData` object that contains information about the columns of the `ResultSet` object that will be returned when this `PreparedStatement` object is executed.

Because a `PreparedStatement` object is precompiled, it is possible to know about the `ResultSet` object that it will return without having to execute it. Consequently, it is possible to invoke the method `getMetaData` on a `PreparedStatement` object rather than waiting to execute it and then invoking the `ResultSet.getMetaData` method on the `ResultSet` object that is returned.

NOTE: Using this method may be expensive for some drivers due to the lack of underlying DBMS support.

Returns:

the description of a `ResultSet` object's columns or `null` if the driver cannot return a `ResultSetMetaData` object

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

setDate

```
public void setDate(int parameterIndex,  
                   Date x,  
                   Calendar cal)  
                throws SQLException
```

Sets the designated parameter to the given `java.sql.Date` value, using the given `Calendar` object. The driver uses the `Calendar` object to construct an SQL DATE value, which the driver then sends to the database. With a `Calendar` object, the driver can calculate the date taking into account a custom timezone. If no `Calendar` object is specified, the driver uses the default timezone, which is that of the virtual machine running the application.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the parameter value

`cal` - the `Calendar` object the driver will use to construct the date

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

setTime

```
public void setTime(int parameterIndex,
                   Time x,
                   Calendar cal)
    throws SQLException
```

Sets the designated parameter to the given `java.sql.Time` value, using the given `Calendar` object. The driver uses the `Calendar` object to construct an SQL TIME value, which the driver then sends to the database. With a `Calendar` object, the driver can calculate the time taking into account a custom timezone. If no `Calendar` object is specified, the driver uses the default timezone, which is that of the virtual machine running the application.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the parameter value

`cal` - the `Calendar` object the driver will use to construct the time

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

setTimestamp

```
public void setTimestamp(int parameterIndex,
```

```

        Timestamp x,
        Calendar cal)
    throws SQLException

```

Sets the designated parameter to the given `java.sql.Timestamp` value, using the given `Calendar` object. The driver uses the `Calendar` object to construct an SQL `TIMESTAMP` value, which the driver then sends to the database. With a `Calendar` object, the driver can calculate the timestamp taking into account a custom timezone. If no `Calendar` object is specified, the driver uses the default timezone, which is that of the virtual machine running the application.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the parameter value

`cal` - the `Calendar` object the driver will use to construct the timestamp

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

setNull

```

public void setNull(int paramIndex,
                    int sqlType,
                    String typeName)
    throws SQLException

```

Sets the designated parameter to SQL `NULL`. This version of the method `setNull` should be used for user-defined types and REF type parameters. Examples of user-defined types include: `STRUCT`, `DISTINCT`, `JAVA_OBJECT`, and named array types.

Note: To be portable, applications must give the SQL type code and the fully-qualified SQL type name when specifying a `NULL` user-defined or REF parameter. In the case of a user-defined type the name is the type name of the parameter itself. For a REF parameter, the name is the type name of the referenced type. If a JDBC driver does not need the type code or type name information, it may ignore it. Although it is intended for user-defined and Ref parameters, this method may be used to set a null parameter of any JDBC type. If the parameter does not have a user-defined or REF type, the given `typeName` is ignored.

Parameters:

`paramIndex` - the first parameter is 1, the second is 2, ...

`sqlType` - a value from `java.sql.Types`

typeName - the fully-qualified name of an SQL user-defined type; ignored if the parameter is not a user-defined type or REF

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

setURL

```
public void setURL(int parameterIndex,  
                    URL x)  
    throws SQLException
```

Sets the designated parameter to the given `java.net.URL` value. The driver converts this to an SQL DATALINK value when it sends it to the database.

Parameters:

`parameterIndex` - the first parameter is 1, the second is 2, ...

`x` - the `java.net.URL` object to be set

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.4

getParameterMetaData

```
public ParameterMetaData getParameterMetaData()  
    throws SQLException
```

Retrieves the number, types and properties of this `PreparedStatement` object's parameters.

Returns:

a `ParameterMetaData` object that contains information about the number, types and properties of this `PreparedStatement` object's parameters

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.4

See Also:

[ParameterMetaData](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

java.sql

Interface ResultSet

All Known Subinterfaces:

[RowSet](#)public interface **ResultSet**

A table of data representing a database result set, which is usually generated by executing a statement that queries the database.

A `ResultSet` object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The `next` method moves the cursor to the next row, and because it returns `false` when there are no more rows in the `ResultSet` object, it can be used in a `while` loop to iterate through the result set.

A default `ResultSet` object is not updatable and has a cursor that moves forward only. Thus, you can iterate through it only once and only from the first row to the last row. It is possible to produce `ResultSet` objects that are scrollable and/or updatable. The following code fragment, in which `con` is a valid `Connection` object, illustrates how to make a result set that is scrollable and insensitive to updates by others, and that is updatable. See `ResultSet` fields for other options.

```
Statement stmt = con.createStatement(
                                ResultSet.
TYPE_SCROLL_INSENSITIVE,
                                ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");
// rs will be scrollable, will not show changes made by
others,
// and will be updatable
```

The `ResultSet` interface provides *getter* methods (`getBoolean`, `getLong`, and so on) for retrieving column values from the current row. Values can be retrieved using either the index number of the column or the name of the column. In general, using the column index will be more efficient.

Columns are numbered from 1. For maximum portability, result set columns within each row should be read in left-to-right order, and each column should be read only once.

For the getter methods, a JDBC driver attempts to convert the underlying data to the Java type specified in the getter method and returns a suitable Java value. The JDBC specification has a table showing the allowable mappings from SQL types to Java types that can be used by the `ResultSet` getter methods.

Column names used as input to getter methods are case insensitive. When a getter method is called with a column name and several columns have the same name, the value of the first matching column will be returned. The column name option is designed to be used when column names are used in the SQL query that generated the result set. For columns that are NOT explicitly named in the query, it is best to use column numbers. If column names are used, there is no way for the programmer to guarantee that they actually refer to the intended columns.

A set of updater methods were added to this interface in the JDBC 2.0 API (JavaTM 2 SDK, Standard Edition, version 1.2). The comments regarding parameters to the getter methods also apply to parameters to the updater methods.

The updater methods may be used in two ways:

1. to update a column value in the current row. In a scrollable `ResultSet` object, the cursor can be moved backwards and forwards, to an absolute position, or to a position relative to the current row. The following code fragment updates the `NAME` column in the fifth row of the `ResultSet` object `rs` and then uses the method `updateRow` to update the data source table from which `rs` was derived.

```
rs.absolute(5); // moves the cursor to the fifth row of
rs
rs.updateString("NAME", "AINSWORTH"); // updates the
    // NAME column of row 5 to be AINSWORTH
rs.updateRow(); // updates the row in the data source
```

2. to insert column values into the insert row. An updatable `ResultSet` object has a special row associated with it that serves as a staging area for building a row to be inserted. The following code fragment moves the cursor to the insert row, builds a three-column row, and inserts it into `rs` and into the data source table using the method `insertRow`.

```
rs.moveToInsertRow(); // moves cursor to the insert row
rs.updateString(1, "AINSWORTH"); // updates the
    // first column of the insert row to be AINSWORTH
```

```

35      rs.updateInt(2,35); // updates the second column to be
      rs.updateBoolean(3, true); // updates the third column
to true
      rs.insertRow();
      rs.moveToCurrentRow();

```

A `ResultSet` object is automatically closed when the `Statement` object that generated it is closed, re-executed, or used to retrieve the next result from a sequence of multiple results.

The number, types and properties of a `ResultSet` object's columns are provided by the `ResultSetMetaData` object returned by the `ResultSet.getMetaData` method.

See Also:

[Statement.executeQuery\(java.lang.String\)](#), [Statement.getResultSet\(\)](#), [ResultSetMetaData](#)

Field Summary

static int	CLOSE_CURSORS_AT_COMMIT The constant indicating that <code>ResultSet</code> objects should be closed when the method <code>Connection.commit</code> is called.
static int	CONCUR_READ_ONLY The constant indicating the concurrency mode for a <code>ResultSet</code> object that may NOT be updated.
static int	CONCUR_UPDATABLE The constant indicating the concurrency mode for a <code>ResultSet</code> object that may be updated.
static int	FETCH_FORWARD The constant indicating that the rows in a result set will be processed in a forward direction; first-to-last.
static int	FETCH_REVERSE The constant indicating that the rows in a result set will be processed in a reverse direction; last-to-first.
static int	FETCH_UNKNOWN The constant indicating that the order in which rows in a result set will be processed is unknown.

static int	HOLD_CURSORS_OVER_COMMIT The constant indicating that <code>ResultSet</code> objects should not be closed when the method <code>Connection.commit</code> is called.
static int	TYPE_FORWARD_ONLY The constant indicating the type for a <code>ResultSet</code> object whose cursor may move only forward.
static int	TYPE_SCROLL_INSENSITIVE The constant indicating the type for a <code>ResultSet</code> object that is scrollable but generally not sensitive to changes made by others.
static int	TYPE_SCROLL_SENSITIVE The constant indicating the type for a <code>ResultSet</code> object that is scrollable and generally sensitive to changes made by others.

Method Summary

boolean	absolute (int row) Moves the cursor to the given row number in this <code>ResultSet</code> object.
void	afterLast () Moves the cursor to the end of this <code>ResultSet</code> object, just after the last row.
void	beforeFirst () Moves the cursor to the front of this <code>ResultSet</code> object, just before the first row.
void	cancelRowUpdates () Cancels the updates made to the current row in this <code>ResultSet</code> object.
void	clearWarnings () Clears all warnings reported on this <code>ResultSet</code> object.
void	close () Releases this <code>ResultSet</code> object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.
void	deleteRow () Deletes the current row from this <code>ResultSet</code> object and from the underlying database.
int	findColumn (String columnName) Maps the given <code>ResultSet</code> column name to its <code>ResultSet</code> column index.

boolean	first() Moves the cursor to the first row in this <code>ResultSet</code> object.
Array	getArray(int i) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as an <code>Array</code> object in the Java programming language.
Array	getArray(String colName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as an <code>Array</code> object in the Java programming language.
InputStream	getAsciiStream(int columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a stream of ASCII characters.
InputStream	getAsciiStream(String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a stream of ASCII characters.
BigDecimal	getBigDecimal(int columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.math.BigDecimal</code> with full precision.
BigDecimal	getBigDecimal(int columnIndex, int scale) Deprecated.
BigDecimal	getBigDecimal(String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.math.BigDecimal</code> with full precision.
BigDecimal	getBigDecimal(String columnName, int scale) Deprecated.
InputStream	getBinaryStream(int columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a binary stream of uninterpreted bytes.
InputStream	getBinaryStream(String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a stream of uninterpreted bytes.
Blob	getBlob(int i) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>Blob</code> object in the Java programming language.

Blob	getBlob (String colName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>Blob</code> object in the Java programming language.
boolean	getBoolean (int columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>boolean</code> in the Java programming language.
boolean	getBoolean (String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>boolean</code> in the Java programming language.
byte	getBytes (int columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>byte</code> in the Java programming language.
byte	getBytes (String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>byte</code> in the Java programming language.
byte[]	getBytes (int columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>byte</code> array in the Java programming language.
byte[]	getBytes (String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>byte</code> array in the Java programming language.
Reader	getCharacterStream (int columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.io.Reader</code> object.
Reader	getCharacterStream (String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.io.Reader</code> object.
Clob	getClob (int i) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>Clob</code> object in the Java programming language.

Clob	getClob (String colName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>Clob</code> object in the Java programming language.
int	getConcurrency () Retrieves the concurrency mode of this <code>ResultSet</code> object.
String	getCursorName () Retrieves the name of the SQL cursor used by this <code>ResultSet</code> object.
Date	getDate (int columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.sql.Date</code> object in the Java programming language.
Date	getDate (int columnIndex, Calendar cal) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.sql.Date</code> object in the Java programming language.
Date	getDate (String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.sql.Date</code> object in the Java programming language.
Date	getDate (String columnName, Calendar cal) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.sql.Date</code> object in the Java programming language.
double	getDouble (int columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>double</code> in the Java programming language.
double	getDouble (String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>double</code> in the Java programming language.
int	getFetchDirection () Retrieves the fetch direction for this <code>ResultSet</code> object.
int	getFetchSize () Retrieves the fetch size for this <code>ResultSet</code> object.
float	getFloat (int columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>float</code> in the Java programming language.

<code>float</code>	<code>getFloat(String columnName)</code> Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>float</code> in the Java programming language.
<code>int</code>	<code>getInt(int columnIndex)</code> Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as an <code>int</code> in the Java programming language.
<code>int</code>	<code>getInt(String columnName)</code> Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as an <code>int</code> in the Java programming language.
<code>long</code>	<code>getLong(int columnIndex)</code> Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>long</code> in the Java programming language.
<code>long</code>	<code>getLong(String columnName)</code> Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>long</code> in the Java programming language.
<code>ResultSetMetaData</code>	<code>getMetaData()</code> Retrieves the number, types and properties of this <code>ResultSet</code> object's columns.
<code>Object</code>	<code>getObject(int columnIndex)</code> Gets the value of the designated column in the current row of this <code>ResultSet</code> object as an <code>Object</code> in the Java programming language.
<code>Object</code>	<code>getObject(int i, Map map)</code> Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as an <code>Object</code> in the Java programming language.
<code>Object</code>	<code>getObject(String columnName)</code> Gets the value of the designated column in the current row of this <code>ResultSet</code> object as an <code>Object</code> in the Java programming language.
<code>Object</code>	<code>getObject(String colName, Map map)</code> Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as an <code>Object</code> in the Java programming language.
<code>Ref</code>	<code>getRef(int i)</code> Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>Ref</code> object in the Java programming language.

Ref	getRef (String colName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>Ref</code> object in the Java programming language.
<code>int</code>	getRow () Retrieves the current row number.
<code>short</code>	getShort (<code>int</code> columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>short</code> in the Java programming language.
<code>short</code>	getShort (String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>short</code> in the Java programming language.
Statement	getStatement () Retrieves the <code>Statement</code> object that produced this <code>ResultSet</code> object.
String	getString (<code>int</code> columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>String</code> in the Java programming language.
String	getString (String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>String</code> in the Java programming language.
Time	getTime (<code>int</code> columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.sql.Time</code> object in the Java programming language.
Time	getTime (<code>int</code> columnIndex, Calendar cal) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.sql.Time</code> object in the Java programming language.
Time	getTime (String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.sql.Time</code> object in the Java programming language.
Time	getTime (String columnName, Calendar cal) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.sql.Time</code> object in the Java programming language.

Timestamp	getTimestamp (int columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.sql.Timestamp</code> object in the Java programming language.
Timestamp	getTimestamp (int columnIndex, Calendar cal) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.sql.Timestamp</code> object in the Java programming language.
Timestamp	getTimestamp (String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.sql.Timestamp</code> object.
Timestamp	getTimestamp (String columnName, Calendar cal) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.sql.Timestamp</code> object in the Java programming language.
int	getType () Retrieves the type of this <code>ResultSet</code> object.
InputStream	getUnicodeStream (int columnIndex) Deprecated. use <i>getCharacterStream</i> in place of <i>getUnicodeStream</i>
InputStream	getUnicodeStream (String columnName) Deprecated. use <i>getCharacterStream</i> instead
URL	getURL (int columnIndex) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.net.URL</code> object in the Java programming language.
URL	getURL (String columnName) Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a <code>java.net.URL</code> object in the Java programming language.
SQLWarning	getWarnings () Retrieves the first warning reported by calls on this <code>ResultSet</code> object.
void	insertRow () Inserts the contents of the insert row into this <code>ResultSet</code> object and into the database.
boolean	isAfterLast () Retrieves whether the cursor is after the last row in this <code>ResultSet</code> object.

boolean	<u>isBeforeFirst()</u> Retrieves whether the cursor is before the first row in this <code>ResultSet</code> object.
boolean	<u>isFirst()</u> Retrieves whether the cursor is on the first row of this <code>ResultSet</code> object.
boolean	<u>isLast()</u> Retrieves whether the cursor is on the last row of this <code>ResultSet</code> object.
boolean	<u>last()</u> Moves the cursor to the last row in this <code>ResultSet</code> object.
void	<u>moveToCurrentRow()</u> Moves the cursor to the remembered cursor position, usually the current row.
void	<u>moveToInsertRow()</u> Moves the cursor to the insert row.
boolean	<u>next()</u> Moves the cursor down one row from its current position.
boolean	<u>previous()</u> Moves the cursor to the previous row in this <code>ResultSet</code> object.
void	<u>refreshRow()</u> Refreshes the current row with its most recent value in the database.
boolean	<u>relative(int rows)</u> Moves the cursor a relative number of rows, either positive or negative.
boolean	<u>rowDeleted()</u> Retrieves whether a row has been deleted.
boolean	<u>rowInserted()</u> Retrieves whether the current row has had an insertion.
boolean	<u>rowUpdated()</u> Retrieves whether the current row has been updated.
void	<u>setFetchDirection(int direction)</u> Gives a hint as to the direction in which the rows in this <code>ResultSet</code> object will be processed.

void	setFetchSize (int rows) Gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed for this <code>ResultSet</code> object.
void	updateArray (int columnIndex, Array x) Updates the designated column with a <code>java.sql.Array</code> value.
void	updateArray (String columnName, Array x) Updates the designated column with a <code>java.sql.Array</code> value.
void	updateAsciiStream (int columnIndex, InputStream x, int length) Updates the designated column with an ascii stream value.
void	updateAsciiStream (String columnName, InputStream x, int length) Updates the designated column with an ascii stream value.
void	updateBigDecimal (int columnIndex, BigDecimal x) Updates the designated column with a <code>java.math.BigDecimal</code> value.
void	updateBigDecimal (String columnName, BigDecimal x) Updates the designated column with a <code>java.sql.BigDecimal</code> value.
void	updateBinaryStream (int columnIndex, InputStream x, int length) Updates the designated column with a binary stream value.
void	updateBinaryStream (String columnName, InputStream x, int length) Updates the designated column with a binary stream value.
void	updateBlob (int columnIndex, Blob x) Updates the designated column with a <code>java.sql.Blob</code> value.
void	updateBlob (String columnName, Blob x) Updates the designated column with a <code>java.sql.Blob</code> value.
void	updateBoolean (int columnIndex, boolean x) Updates the designated column with a boolean value.
void	updateBoolean (String columnName, boolean x) Updates the designated column with a boolean value.
void	updateByte (int columnIndex, byte x) Updates the designated column with a byte value.

void	updateByte (String columnName, byte x) Updates the designated column with a byte value.
void	updateBytes (int columnIndex, byte[] x) Updates the designated column with a byte array value.
void	updateBytes (String columnName, byte[] x) Updates the designated column with a byte array value.
void	updateCharacterStream (int columnIndex, Reader x, int length) Updates the designated column with a character stream value.
void	updateCharacterStream (String columnName, Reader reader, int length) Updates the designated column with a character stream value.
void	updateClob (int columnIndex, Clob x) Updates the designated column with a java.sql.Clob value.
void	updateClob (String columnName, Clob x) Updates the designated column with a java.sql.Clob value.
void	updateDate (int columnIndex, Date x) Updates the designated column with a java.sql.Date value.
void	updateDate (String columnName, Date x) Updates the designated column with a java.sql.Date value.
void	updateDouble (int columnIndex, double x) Updates the designated column with a double value.
void	updateDouble (String columnName, double x) Updates the designated column with a double value.
void	updateFloat (int columnIndex, float x) Updates the designated column with a float value.
void	updateFloat (String columnName, float x) Updates the designated column with a float value.
void	updateInt (int columnIndex, int x) Updates the designated column with an int value.
void	updateInt (String columnName, int x) Updates the designated column with an int value.
void	updateLong (int columnIndex, long x) Updates the designated column with a long value.
void	updateLong (String columnName, long x) Updates the designated column with a long value.

void	updateNull (int columnIndex) Gives a nullable column a null value.
void	updateNull (String columnName) Updates the designated column with a null value.
void	updateObject (int columnIndex, Object x) Updates the designated column with an Object value.
void	updateObject (int columnIndex, Object x, int scale) Updates the designated column with an Object value.
void	updateObject (String columnName, Object x) Updates the designated column with an Object value.
void	updateObject (String columnName, Object x, int scale) Updates the designated column with an Object value.
void	updateRef (int columnIndex, Ref x) Updates the designated column with a java.sql.Ref value.
void	updateRef (String columnName, Ref x) Updates the designated column with a java.sql.Ref value.
void	updateRow () Updates the underlying database with the new contents of the current row of this ResultSet object.
void	updateShort (int columnIndex, short x) Updates the designated column with a short value.
void	updateShort (String columnName, short x) Updates the designated column with a short value.
void	updateString (int columnIndex, String x) Updates the designated column with a String value.
void	updateString (String columnName, String x) Updates the designated column with a String value.
void	updateTime (int columnIndex, Time x) Updates the designated column with a java.sql.Time value.
void	updateTime (String columnName, Time x) Updates the designated column with a java.sql.Time value.
void	updateTimestamp (int columnIndex, Timestamp x) Updates the designated column with a java.sql.Timestamp value.

void	updateTimestamp (String columnName, Timestamp x) Updates the designated column with a <code>java.sql.Timestamp</code> value.
boolean	wasNull () Reports whether the last column read had a value of SQL NULL.

Field Detail

FETCH_FORWARD

```
public static final int FETCH_FORWARD
```

The constant indicating that the rows in a result set will be processed in a forward direction; first-to-last. This constant is used by the method `setFetchDirection` as a hint to the driver, which the driver may ignore.

Since:

1.2

See Also:

[Constant Field Values](#)

FETCH_REVERSE

```
public static final int FETCH_REVERSE
```

The constant indicating that the rows in a result set will be processed in a reverse direction; last-to-first. This constant is used by the method `setFetchDirection` as a hint to the driver, which the driver may ignore.

Since:

1.2

See Also:

[Constant Field Values](#)

FETCH_UNKNOWN

```
public static final int FETCH_UNKNOWN
```

The constant indicating that the order in which rows in a result set will be processed is unknown. This constant is used by the method `setFetchDirection` as a hint to the driver, which the driver may ignore.

See Also:

[Constant Field Values](#)

TYPE_FORWARD_ONLY

```
public static final int TYPE_FORWARD_ONLY
```

The constant indicating the type for a `ResultSet` object whose cursor may move only forward.

Since:

1.2

See Also:

[Constant Field Values](#)

TYPE_SCROLL_INSENSITIVE

```
public static final int TYPE_SCROLL_INSENSITIVE
```

The constant indicating the type for a `ResultSet` object that is scrollable but generally not sensitive to changes made by others.

Since:

1.2

See Also:

[Constant Field Values](#)

TYPE_SCROLL_SENSITIVE

```
public static final int TYPE_SCROLL_SENSITIVE
```

The constant indicating the type for a `ResultSet` object that is scrollable and generally

sensitive to changes made by others.

Since:

1.2

See Also:

[Constant Field Values](#)

CONCUR_READ_ONLY

```
public static final int CONCUR_READ_ONLY
```

The constant indicating the concurrency mode for a `ResultSet` object that may NOT be updated.

Since:

1.2

See Also:

[Constant Field Values](#)

CONCUR_UPDATABLE

```
public static final int CONCUR_UPDATABLE
```

The constant indicating the concurrency mode for a `ResultSet` object that may be updated.

Since:

1.2

See Also:

[Constant Field Values](#)

HOLD_CURSORS_OVER_COMMIT

```
public static final int HOLD_CURSORS_OVER_COMMIT
```

The constant indicating that `ResultSet` objects should not be closed when the method `Connection.commit` is called.

Since:

1.4

See Also:[Constant Field Values](#)

CLOSE_CURSORS_AT_COMMIT

```
public static final int CLOSE_CURSORS_AT_COMMIT
```

The constant indicating that `ResultSet` objects should be closed when the method `Connection.commit` is called.

Since:

1.4

See Also:[Constant Field Values](#)

Method Detail

next

```
public boolean next()  
    throws SQLException
```

Moves the cursor down one row from its current position. A `ResultSet` cursor is initially positioned before the first row; the first call to the method `next` makes the first row the current row; the second call makes the second row the current row, and so on.

If an input stream is open for the current row, a call to the method `next` will implicitly close it. A `ResultSet` object's warning chain is cleared when a new row is read.

Returns:

`true` if the new current row is valid; `false` if there are no more rows

Throws:

[SQLException](#) - if a database access error occurs

close

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

Releases this `ResultSet` object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.

Note: A `ResultSet` object is automatically closed by the `Statement` object that generated it when that `Statement` object is closed, re-executed, or is used to retrieve the next result from a sequence of multiple results. A `ResultSet` object is also automatically closed when it is garbage collected.

Throws:

[SQLException](#) - if a database access error occurs

wasNull

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

Reports whether the last column read had a value of SQL NULL. Note that you must first call one of the getter methods on a column to try to read its value and then call the method `wasNull` to see if the value read was SQL NULL.

Returns:

`true` if the last column value read was SQL NULL and `false` otherwise

Throws:

[SQLException](#) - if a database access error occurs

getString

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

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `String` in the Java programming language.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

the column value; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getBoolean

```
public boolean getBoolean(int columnIndex)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `boolean` in the Java programming language.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

the column value; if the value is SQL NULL, the value returned is `false`

Throws:

[SQLException](#) - if a database access error occurs

getBytes

```
public byte getBytes(int columnIndex)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `byte` in the Java programming language.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

the column value; if the value is SQL NULL, the value returned is `0`

Throws:

[SQLException](#) - if a database access error occurs

getShort

```
public short getShort(int columnIndex)
```

throws [SQLException](#)

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `short` in the Java programming language.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

the column value; if the value is SQL NULL, the value returned is 0

Throws:

[SQLException](#) - if a database access error occurs

getInt

```
public int getInt(int columnIndex)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as an `int` in the Java programming language.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

the column value; if the value is SQL NULL, the value returned is 0

Throws:

[SQLException](#) - if a database access error occurs

getLong

```
public long getLong(int columnIndex)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `long` in the Java programming language.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

the column value; if the value is SQL NULL, the value returned is 0

Throws:

[SQLException](#) - if a database access error occurs

getFloat

```
public float getFloat(int columnIndex)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `float` in the Java programming language.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

the column value; if the value is SQL NULL, the value returned is 0

Throws:

[SQLException](#) - if a database access error occurs

getDouble

```
public double getDouble(int columnIndex)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `double` in the Java programming language.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

the column value; if the value is SQL NULL, the value returned is 0

Throws:

[SQLException](#) - if a database access error occurs

getBigDecimal

```
public BigDecimal getBigDecimal(int columnIndex,  
                                   int scale)  
    throws SQLException
```

Deprecated.

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.BigDecimal` in the Java programming language.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

`scale` - the number of digits to the right of the decimal point

Returns:

the column value; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getBytes

```
public byte[] getBytes(int columnIndex)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `byte` array in the Java programming language. The bytes represent the raw values returned by the driver.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

the column value; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getDate

```
public Date getDate(int columnIndex)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date` object in the Java programming language.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

the column value; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getTime

```
public Time getTime(int columnIndex)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Time` object in the Java programming language.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

the column value; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getTimestamp

```
public Timestamp getTimestamp(int columnIndex)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Timestamp` object in the Java programming language.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

the column value; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getAsciiStream

```
public InputStream getAsciiStream(int columnIndex)
        throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a stream of ASCII characters. The value can then be read in chunks from the stream. This method is particularly suitable for retrieving large `LONGVARCHAR` values. The JDBC driver will do any necessary conversion from the database format into ASCII.

Note: All the data in the returned stream must be read prior to getting the value of any other column. The next call to a getter method implicitly closes the stream. Also, a stream may return 0 when the method `InputStream.available` is called whether there is data available or not.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

a Java input stream that delivers the database column value as a stream of one-byte ASCII characters; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getUnicodeStream

```
public InputStream getUnicodeStream(int columnIndex)
        throws SQLException
```

Deprecated. *use `getCharacterStream` in place of `getUnicodeStream`*

Retrieves the value of the designated column in the current row of this `ResultSet` object as a stream of two-byte Unicode characters. The first byte is the high byte; the second byte is the low byte. The value can then be read in chunks from the stream. This method is particularly suitable for retrieving large `LONGVARCHAR` values. The JDBC driver will do any necessary conversion from the database format into Unicode.

Note: All the data in the returned stream must be read prior to getting the value of any other column. The next call to a getter method implicitly closes the stream. Also, a stream may return 0 when the method `InputStream.available` is called, whether there is data available or not.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

a Java input stream that delivers the database column value as a stream of two-byte Unicode characters; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getBinaryStream

```
public InputStream getBinaryStream(int columnIndex)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a binary stream of uninterpreted bytes. The value can then be read in chunks from the stream. This method is particularly suitable for retrieving large `LONGVARBINARY` values.

Note: All the data in the returned stream must be read prior to getting the value of any other column. The next call to a getter method implicitly closes the stream. Also, a stream may return 0 when the method `InputStream.available` is called whether there is data available or not.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

a Java input stream that delivers the database column value as a stream of uninterpreted bytes; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getString

```
public String getString(String columnName)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `String` in the Java programming language.

Parameters:

`columnName` - the SQL name of the column

Returns:

the column value; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getBoolean

```
public boolean getBoolean(String columnName)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `boolean` in the Java programming language.

Parameters:

columnName - the SQL name of the column

Returns:

the column value; if the value is SQL NULL, the value returned is `false`

Throws:

[SQLException](#) - if a database access error occurs

getBytes

```
public byte getBytes(String columnName)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `byte` in the Java programming language.

Parameters:

columnName - the SQL name of the column

Returns:

the column value; if the value is SQL NULL, the value returned is `0`

Throws:

[SQLException](#) - if a database access error occurs

getShort

```
public short getShort(String columnName)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `short` in the Java programming language.

Parameters:

`columnName` - the SQL name of the column

Returns:

the column value; if the value is SQL NULL, the value returned is 0

Throws:

[SQLException](#) - if a database access error occurs

getInt

```
public int getInt(String columnName)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as an `int` in the Java programming language.

Parameters:

`columnName` - the SQL name of the column

Returns:

the column value; if the value is SQL NULL, the value returned is 0

Throws:

[SQLException](#) - if a database access error occurs

getLong

```
public long getLong(String columnName)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `long` in the Java programming language.

Parameters:

`columnName` - the SQL name of the column

Returns:

the column value; if the value is SQL NULL, the value returned is 0

Throws:

[SQLException](#) - if a database access error occurs

getFloat

```
public float getFloat(String columnName)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `float` in the Java programming language.

Parameters:

columnName - the SQL name of the column

Returns:

the column value; if the value is SQL NULL, the value returned is 0

Throws:

[SQLException](#) - if a database access error occurs

getDouble

```
public double getDouble(String columnName)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `double` in the Java programming language.

Parameters:

columnName - the SQL name of the column

Returns:

the column value; if the value is SQL NULL, the value returned is 0

Throws:

[SQLException](#) - if a database access error occurs

getBigDecimal

```
public BigDecimal getBigDecimal(String columnName,
    int scale)
    throws SQLException
```

Deprecated.

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.math.BigDecimal` in the Java programming language.

Parameters:

`columnName` - the SQL name of the column

`scale` - the number of digits to the right of the decimal point

Returns:

the column value; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getBytes

```
public byte[] getBytes(String columnName)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `byte` array in the Java programming language. The bytes represent the raw values returned by the driver.

Parameters:

`columnName` - the SQL name of the column

Returns:

the column value; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getDate

```
public Date getDate(String columnName)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date` object in the Java programming language.

Parameters:

`columnName` - the SQL name of the column

Returns:

the column value; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getTime

```
public Time getTime(String columnName)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Time` object in the Java programming language.

Parameters:

columnName - the SQL name of the column

Returns:

the column value; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getTimestamp

```
public Timestamp getTimestamp(String columnName)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Timestamp` object.

Parameters:

columnName - the SQL name of the column

Returns:

the column value; if the value is SQL NULL, the value returned is `null`

Throws:

[SQLException](#) - if a database access error occurs

getAsciiStream

```
public InputStream getAsciiStream(String columnName)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a stream of ASCII characters. The value can then be read in chunks from the stream. This method is particularly suitable for retrieving large `LONGVARCHAR` values. The JDBC driver will do any necessary conversion from the database format into ASCII.

Note: All the data in the returned stream must be read prior to getting the value of any other column. The next call to a getter method implicitly closes the stream. Also, a stream may return 0 when the method `available` is called whether there is data available or not.

Parameters:

`columnName` - the SQL name of the column

Returns:

a Java input stream that delivers the database column value as a stream of one-byte ASCII characters. If the value is SQL NULL, the value returned is `null`.

Throws:

[SQLException](#) - if a database access error occurs

getUnicodeStream

```
public InputStream getUnicodeStream(String columnName)
                                throws SQLException
```

Deprecated. *use `getCharacterStream` instead*

Retrieves the value of the designated column in the current row of this `ResultSet` object as a stream of two-byte Unicode characters. The first byte is the high byte; the second byte is the low byte. The value can then be read in chunks from the stream. This method is particularly suitable for retrieving large `LONGVARCHAR` values. The JDBC technology-enabled driver will do any necessary conversion from the database format into Unicode.

Note: All the data in the returned stream must be read prior to getting the value of any other column. The next call to a getter method implicitly closes the stream. Also, a stream may return 0 when the method `InputStream.available` is called, whether there is data available or not.

Parameters:

`columnName` - the SQL name of the column

Returns:

a Java input stream that delivers the database column value as a stream of two-byte Unicode characters. If the value is SQL NULL, the value returned is `null`.

Throws:

[SQLException](#) - if a database access error occurs

getBinaryStream

```
public InputStream getBinaryStream(String columnName)  
                                throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a stream of uninterpreted bytes. The value can then be read in chunks from the stream. This method is particularly suitable for retrieving large `LONGVARBINARY` values.

Note: All the data in the returned stream must be read prior to getting the value of any other column. The next call to a getter method implicitly closes the stream. Also, a stream may return `0` when the method `available` is called whether there is data available or not.

Parameters:

columnName - the SQL name of the column

Returns:

a Java input stream that delivers the database column value as a stream of uninterpreted bytes; if the value is SQL NULL, the result is `null`

Throws:

[SQLException](#) - if a database access error occurs

getWarnings

```
public SQLWarning getWarnings()  
                                throws SQLException
```

Retrieves the first warning reported by calls on this `ResultSet` object. Subsequent warnings on this `ResultSet` object will be chained to the `SQLWarning` object that this method returns.

The warning chain is automatically cleared each time a new row is read. This method may not be called on a `ResultSet` object that has been closed; doing so will cause an `SQLException` to be thrown.

Note: This warning chain only covers warnings caused by `ResultSet` methods. Any warning caused by `Statement` methods (such as reading OUT parameters) will be chained on the `Statement` object.

Returns:

the first `SQLWarning` object reported or `null` if there are none

Throws:

[SQLException](#) - if a database access error occurs or this method is called on a closed result set

clearWarnings

```
public void clearWarnings()  
           throws SQLException
```

Clears all warnings reported on this `ResultSet` object. After this method is called, the method `getWarnings` returns `null` until a new warning is reported for this `ResultSet` object.

Throws:

[SQLException](#) - if a database access error occurs

getCursorName

```
public String getCursorName()  
           throws SQLException
```

Retrieves the name of the SQL cursor used by this `ResultSet` object.

In SQL, a result table is retrieved through a cursor that is named. The current row of a result set can be updated or deleted using a positioned update/delete statement that references the cursor name. To insure that the cursor has the proper isolation level to support update, the cursor's `SELECT` statement should be of the form `SELECT FOR UPDATE`. If `FOR UPDATE` is omitted, the positioned updates may fail.

The JDBC API supports this SQL feature by providing the name of the SQL cursor used by a `ResultSet` object. The current row of a `ResultSet` object is also the current row of this SQL cursor.

Note: If positioned update is not supported, a `SQLException` is thrown.

Returns:

the SQL name for this `ResultSet` object's cursor

Throws:

[SQLException](#) - if a database access error occurs

getMetaData

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

Retrieves the number, types and properties of this `ResultSet` object's columns.

Returns:

the description of this `ResultSet` object's columns

Throws:

[SQLException](#) - if a database access error occurs

getObject

```
public Object getObject(int columnIndex)  
                        throws SQLException
```

Gets the value of the designated column in the current row of this `ResultSet` object as an `Object` in the Java programming language.

This method will return the value of the given column as a Java object. The type of the Java object will be the default Java object type corresponding to the column's SQL type, following the mapping for built-in types specified in the JDBC specification. If the value is an SQL `NULL`, the driver returns a Java `null`.

This method may also be used to read database-specific abstract data types. In the JDBC 2.0 API, the behavior of method `getObject` is extended to materialize data of SQL user-defined types. When a column contains a structured or distinct value, the behavior of this method is as if it were a call to: `getObject(columnIndex, this.getStatement().getConnection().getTypeMap())`.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

a `java.lang.Object` holding the column value

Throws:

[SQLException](#) - if a database access error occurs

getObject

```
public Object getObject(String columnName)  
    throws SQLException
```

Gets the value of the designated column in the current row of this `ResultSet` object as an `Object` in the Java programming language.

This method will return the value of the given column as a Java object. The type of the Java object will be the default Java object type corresponding to the column's SQL type, following the mapping for built-in types specified in the JDBC specification. If the value is an SQL `NULL`, the driver returns a Java `null`.

This method may also be used to read database-specific abstract data types.

In the JDBC 2.0 API, the behavior of the method `getObject` is extended to materialize data of SQL user-defined types. When a column contains a structured or distinct value, the behavior of this method is as if it were a call to: `getObject(columnIndex, this.getStatement().getConnection().getTypeMap())`.

Parameters:

`columnName` - the SQL name of the column

Returns:

a `java.lang.Object` holding the column value

Throws:

[SQLException](#) - if a database access error occurs

findColumn

```
public int findColumn(String columnName)  
    throws SQLException
```

Maps the given `ResultSet` column name to its `ResultSet` column index.

Parameters:

`columnName` - the name of the column

Returns:

the column index of the given column name

Throws:

[SQLException](#) - if the `ResultSet` object does not contain `columnName` or a database access error occurs

getCharacterStream

```
public Reader getCharacterStream(int columnIndex)
                               throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.io.Reader` object.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

a `java.io.Reader` object that contains the column value; if the value is SQL `NULL`, the value returned is `null` in the Java programming language.

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getCharacterStream

```
public Reader getCharacterStream(String columnName)
                               throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.io.Reader` object.

Parameters:

`columnName` - the name of the column

Returns:

a `java.io.Reader` object that contains the column value; if the value is SQL `NULL`, the value returned is `null` in the Java programming language

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getBigDecimal

```
public BigDecimal getBigDecimal(int columnIndex)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.math.BigDecimal` with full precision.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Returns:

the column value (full precision); if the value is SQL NULL, the value returned is `null` in the Java programming language.

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getBigDecimal

```
public BigDecimal getBigDecimal(String columnName)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.math.BigDecimal` with full precision.

Parameters:

`columnName` - the column name

Returns:

the column value (full precision); if the value is SQL NULL, the value returned is `null` in the Java programming language.

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

isBeforeFirst

```
public boolean isBeforeFirst()  
                throws SQLException
```

Retrieves whether the cursor is before the first row in this `ResultSet` object.

Returns:

`true` if the cursor is before the first row; `false` if the cursor is at any other position or the result set contains no rows

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

isAfterLast

```
public boolean isAfterLast()  
                throws SQLException
```

Retrieves whether the cursor is after the last row in this `ResultSet` object.

Returns:

`true` if the cursor is after the last row; `false` if the cursor is at any other position or the result set contains no rows

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

isFirst

```
public boolean isFirst()  
                throws SQLException
```

Retrieves whether the cursor is on the first row of this `ResultSet` object.

Returns:

`true` if the cursor is on the first row; `false` otherwise

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

isLast

```
public boolean isLast()  
    throws SQLException
```

Retrieves whether the cursor is on the last row of this `ResultSet` object. Note: Calling the method `isLast` may be expensive because the JDBC driver might need to fetch ahead one row in order to determine whether the current row is the last row in the result set.

Returns:

`true` if the cursor is on the last row; `false` otherwise

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

beforeFirst

```
public void beforeFirst()  
    throws SQLException
```

Moves the cursor to the front of this `ResultSet` object, just before the first row. This method has no effect if the result set contains no rows.

Throws:

[SQLException](#) - if a database access error occurs or the result set type is `TYPE_FORWARD_ONLY`

Since:

1.2

afterLast

```
public void afterLast()
```

throws [SQLException](#)

Moves the cursor to the end of this `ResultSet` object, just after the last row. This method has no effect if the result set contains no rows.

Throws:

[SQLException](#) - if a database access error occurs or the result set type is `TYPE_FORWARD_ONLY`

Since:

1.2

first

```
public boolean first()  
    throws SQLException
```

Moves the cursor to the first row in this `ResultSet` object.

Returns:

`true` if the cursor is on a valid row; `false` if there are no rows in the result set

Throws:

[SQLException](#) - if a database access error occurs or the result set type is `TYPE_FORWARD_ONLY`

Since:

1.2

last

```
public boolean last()  
    throws SQLException
```

Moves the cursor to the last row in this `ResultSet` object.

Returns:

`true` if the cursor is on a valid row; `false` if there are no rows in the result set

Throws:

[SQLException](#) - if a database access error occurs or the result set type is `TYPE_FORWARD_ONLY`

Since:

1.2

getRow

```
public int getRow()  
    throws SQLException
```

Retrieves the current row number. The first row is number 1, the second number 2, and so on.

Returns:

the current row number; 0 if there is no current row

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

absolute

```
public boolean absolute(int row)  
    throws SQLException
```

Moves the cursor to the given row number in this `ResultSet` object.

If the row number is positive, the cursor moves to the given row number with respect to the beginning of the result set. The first row is row 1, the second is row 2, and so on.

If the given row number is negative, the cursor moves to an absolute row position with respect to the end of the result set. For example, calling the method `absolute(-1)` positions the cursor on the last row; calling the method `absolute(-2)` moves the cursor to the next-to-last row, and so on.

An attempt to position the cursor beyond the first/last row in the result set leaves the cursor before the first row or after the last row.

Note: Calling `absolute(1)` is the same as calling `first()`. Calling `absolute(-1)` is the same as calling `last()`.

Parameters:

`row` - the number of the row to which the cursor should move. A positive number indicates the row number counting from the beginning of the result set; a negative number indicates the row number counting from the end of the result set

Returns:

`true` if the cursor is on the result set; `false` otherwise

Throws:

[SQLException](#) - if a database access error occurs, or the result set type is `TYPE_FORWARD_ONLY`

Since:

1.2

relative

```
public boolean relative(int rows)
    throws SQLException
```

Moves the cursor a relative number of rows, either positive or negative. Attempting to move beyond the first/last row in the result set positions the cursor before/after the the first/last row. Calling `relative(0)` is valid, but does not change the cursor position.

Note: Calling the method `relative(1)` is identical to calling the method `next()` and calling the method `relative(-1)` is identical to calling the method `previous()`.

Parameters:

`rows` - an `int` specifying the number of rows to move from the current row; a positive number moves the cursor forward; a negative number moves the cursor backward

Returns:

`true` if the cursor is on a row; `false` otherwise

Throws:

[SQLException](#) - if a database access error occurs, there is no current row, or the result set type is `TYPE_FORWARD_ONLY`

Since:

1.2

previous

```
public boolean previous()
    throws SQLException
```

Moves the cursor to the previous row in this `ResultSet` object.

Returns:

true if the cursor is on a valid row; false if it is off the result set

Throws:

[SQLException](#) - if a database access error occurs or the result set type is
TYPE_FORWARD_ONLY

Since:

1.2

setFetchDirection

```
public void setFetchDirection(int direction)
           throws SQLException
```

Gives a hint as to the direction in which the rows in this `ResultSet` object will be processed. The initial value is determined by the `Statement` object that produced this `ResultSet` object. The fetch direction may be changed at any time.

Parameters:

`direction` - an `int` specifying the suggested fetch direction; one of `ResultSet.FETCH_FORWARD`, `ResultSet.FETCH_REVERSE`, or `ResultSet.FETCH_UNKNOWN`

Throws:

[SQLException](#) - if a database access error occurs or the result set type is
TYPE_FORWARD_ONLY and the fetch direction is not `FETCH_FORWARD`

Since:

1.2

See Also:

[Statement.setFetchDirection\(int\)](#), [getFetchDirection\(\)](#)

getFetchDirection

```
public int getFetchDirection()
           throws SQLException
```

Retrieves the fetch direction for this `ResultSet` object.

Returns:

the current fetch direction for this `ResultSet` object

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

See Also:[setFetchDirection\(int\)](#)

setFetchSize

```
public void setFetchSize(int rows)
    throws SQLException
```

Gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed for this `ResultSet` object. If the fetch size specified is zero, the JDBC driver ignores the value and is free to make its own best guess as to what the fetch size should be. The default value is set by the `Statement` object that created the result set. The fetch size may be changed at any time.

Parameters:

`rows` - the number of rows to fetch

Throws:

[SQLException](#) - if a database access error occurs or the condition `0 <= rows <= Statement.getMaxRows()` is not satisfied

Since:

1.2

See Also:[getFetchSize\(\)](#)

getFetchSize

```
public int getFetchSize()
    throws SQLException
```

Retrieves the fetch size for this `ResultSet` object.

Returns:

the current fetch size for this `ResultSet` object

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

See Also:[setFetchSize\(int\)](#)

getType

```
public int getType()  
    throws SQLException
```

Retrieves the type of this `ResultSet` object. The type is determined by the `Statement` object that created the result set.

Returns:

`ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, or `ResultSet.TYPE_SCROLL_SENSITIVE`

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getConcurrency

```
public int getConcurrency()  
    throws SQLException
```

Retrieves the concurrency mode of this `ResultSet` object. The concurrency used is determined by the `Statement` object that created the result set.

Returns:

the concurrency type, either `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

rowUpdated

```
public boolean rowUpdated()  
    throws SQLException
```

Retrieves whether the current row has been updated. The value returned depends on whether or not the result set can detect updates.

Returns:

true if both (1) the row has been visibly updated by the owner or another and (2) updates are detected

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

See Also:

[DatabaseMetaData.updateAreDetected\(int\)](#)

rowInserted

```
public boolean rowInserted()  
    throws SQLException
```

Retrieves whether the current row has had an insertion. The value returned depends on whether or not this `ResultSet` object can detect visible inserts.

Returns:

true if a row has had an insertion and insertions are detected; false otherwise

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

See Also:

[DatabaseMetaData.insertsAreDetected\(int\)](#)

rowDeleted

```
public boolean rowDeleted()  
    throws SQLException
```

Retrieves whether a row has been deleted. A deleted row may leave a visible "hole" in a result set. This method can be used to detect holes in a result set. The value returned depends on whether or not this `ResultSet` object can detect deletions.

Returns:

`true` if a row was deleted and deletions are detected; `false` otherwise

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

See Also:

[DatabaseMetaData.deletesAreDetected\(int\)](#)

updateNull

```
public void updateNull(int columnIndex)
    throws SQLException
```

Gives a nullable column a null value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateBoolean

```
public void updateBoolean(int columnIndex,
    boolean x)
    throws SQLException
```

Updates the designated column with a `boolean` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateByte

```
public void updateByte(int columnIndex,  
                        byte x)  
    throws SQLException
```

Updates the designated column with a `byte` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateShort

```
public void updateShort(int columnIndex,  
                        short x)  
    throws SQLException
```

Updates the designated column with a `short` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateInt

```
public void updateInt(int columnIndex,  
                      int x)  
    throws SQLException
```

Updates the designated column with an `int` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateLong

```
public void updateLong(int columnIndex,  
                       long x)  
    throws SQLException
```

Updates the designated column with a `long` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateFloat

```
public void updateFloat(int columnIndex,  
                        float x)  
    throws SQLException
```

Updates the designated column with a `float` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateDouble

```
public void updateDouble(int columnIndex,  
                          double x)  
    throws SQLException
```

Updates the designated column with a `double` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateBigDecimal

```
public void updateBigDecimal(int columnIndex,  
                               BigDecimal x)  
    throws SQLException
```

Updates the designated column with a `java.math.BigDecimal` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateString

```
public void updateString(int columnIndex,  
                           String x)  
    throws SQLException
```

Updates the designated column with a `String` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateBytes

```
public void updateBytes(int columnIndex,
                        byte[] x)
    throws SQLException
```

Updates the designated column with a `byte` array value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateDate

```
public void updateDate(int columnIndex,
                        Date x)
    throws SQLException
```

Updates the designated column with a `java.sql.Date` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateTime

```
public void updateTime(int columnIndex,
                        Time x)
```

throws [SQLException](#)

Updates the designated column with a `java.sql.Time` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateTimestamp

```
public void updateTimestamp(int columnIndex,
                             Timestamp x)
    throws SQLException
```

Updates the designated column with a `java.sql.Timestamp` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateAsciiStream

```
public void updateAsciiStream(int columnIndex,
                               InputStream x,
                               int length)
    throws SQLException
```

Updates the designated column with an ascii stream value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

`x` - the new column value

`length` - the length of the stream

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateBinaryStream

```
public void updateBinaryStream(int columnIndex,  
                                InputStream x,  
                                int length)  
    throws SQLException
```

Updates the designated column with a binary stream value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

`x` - the new column value

`length` - the length of the stream

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateCharacterStream

```
public void updateCharacterStream(int columnIndex,  
                                   Reader x,
```

`int length)`
throws [SQLException](#)

Updates the designated column with a character stream value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

`x` - the new column value

`length` - the length of the stream

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateObject

```
public void updateObject(int columnIndex,  
                           Object x,  
                           int scale)  
    throws SQLException
```

Updates the designated column with an `Object` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

`x` - the new column value

`scale` - for `java.sql.Types.DECIMA` or `java.sql.Types.NUMERIC` types, this is the number of digits after the decimal point. For all other types this value will be ignored.

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateObject

```
public void updateObject(int columnIndex,  
                           Object x)  
    throws SQLException
```

Updates the designated column with an `Object` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateNull

```
public void updateNull(String columnName)  
    throws SQLException
```

Updates the designated column with a `null` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateBoolean

```
public void updateBoolean(String columnName,
```


boolean x)
throws [SQLException](#)

Updates the designated column with a `boolean` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

columnName - the name of the column
x - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateByte

```
public void updateByte(String columnName,  
                       byte x)  
    throws SQLException
```

Updates the designated column with a `byte` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

columnName - the name of the column
x - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateShort

```
public void updateShort(String columnName,  
                       short x)  
    throws SQLException
```

Updates the designated column with a `short` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateInt

```
public void updateInt(String columnName,  
                     int x)  
    throws SQLException
```

Updates the designated column with an `int` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateLong

```
public void updateLong(String columnName,  
                      long x)  
    throws SQLException
```

Updates the designated column with a `long` value. The updater methods are used to update

column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateFloat

```
public void updateFloat(String columnName,  
                        float x)  
    throws SQLException
```

Updates the designated column with a `float` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateDouble

```
public void updateDouble(String columnName,  
                        double x)  
    throws SQLException
```

Updates the designated column with a `double` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update

the database.

Parameters:

`columnName` - the name of the column

`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateBigDecimal

```
public void updateBigDecimal(String columnName,  
                               BigDecimal x)  
    throws SQLException
```

Updates the designated column with a `java.sql.BigDecimal` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column

`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateString

```
public void updateString(String columnName,  
                          String x)  
    throws SQLException
```

Updates the designated column with a `String` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

columnName - the name of the column

x - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateBytes

```
public void updateBytes(String columnName,  
                        byte[] x)  
    throws SQLException
```

Updates the designated column with a byte array value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

columnName - the name of the column

x - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateDate

```
public void updateDate(String columnName,  
                       Date x)  
    throws SQLException
```

Updates the designated column with a `java.sql.Date` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

columnName - the name of the column

x - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateTime

```
public void updateTime(String columnName,  
                      Time x)  
    throws SQLException
```

Updates the designated column with a `java.sql.Time` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

columnName - the name of the column

x - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateTimestamp

```
public void updateTimestamp(String columnName,  
                           Timestamp x)  
    throws SQLException
```

Updates the designated column with a `java.sql.Timestamp` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

columnName - the name of the column

x - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateAsciiStream

```
public void updateAsciiStream(String columnName,  
                               InputStream x,  
                               int length)  
    throws SQLException
```

Updates the designated column with an ascii stream value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column

`x` - the new column value

`length` - the length of the stream

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateBinaryStream

```
public void updateBinaryStream(String columnName,  
                                InputStream x,  
                                int length)  
    throws SQLException
```

Updates the designated column with a binary stream value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column

x - the new column value
length - the length of the stream

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateCharacterStream

```
public void updateCharacterStream(String columnName,  
                                Reader reader,  
                                int length)  
    throws SQLException
```

Updates the designated column with a character stream value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

columnName - the name of the column
reader - the `java.io.Reader` object containing the new column value
length - the length of the stream

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateObject

```
public void updateObject(String columnName,  
                        Object x,  
                        int scale)  
    throws SQLException
```

Updates the designated column with an `Object` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column

`x` - the new column value

`scale` - for `java.sql.Types.DECIMAL` or `java.sql.Types.NUMERIC` types, this is the number of digits after the decimal point. For all other types this value will be ignored.

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

updateObject

```
public void updateObject(String columnName,  
                        Object x)  
    throws SQLException
```

Updates the designated column with an `Object` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column

`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

insertRow

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

Inserts the contents of the insert row into this `ResultSet` object and into the database. The cursor must be on the insert row when this method is called.

Throws:

[SQLException](#) - if a database access error occurs, if this method is called when the cursor is not on the insert row, or if not all of non-nullable columns in the insert row have been given a value

Since:

1.2

updateRow

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

Updates the underlying database with the new contents of the current row of this `ResultSet` object. This method cannot be called when the cursor is on the insert row.

Throws:

[SQLException](#) - if a database access error occurs or if this method is called when the cursor is on the insert row

Since:

1.2

deleteRow

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

Deletes the current row from this `ResultSet` object and from the underlying database. This method cannot be called when the cursor is on the insert row.

Throws:

[SQLException](#) - if a database access error occurs or if this method is called when the cursor is on the insert row

Since:

1.2

refreshRow

```
public void refreshRow()
```

throws [SQLException](#)

Refreshes the current row with its most recent value in the database. This method cannot be called when the cursor is on the insert row.

The `refreshRow` method provides a way for an application to explicitly tell the JDBC driver to refetch a row(s) from the database. An application may want to call `refreshRow` when caching or prefetching is being done by the JDBC driver to fetch the latest value of a row from the database. The JDBC driver may actually refresh multiple rows at once if the fetch size is greater than one.

All values are refetched subject to the transaction isolation level and cursor sensitivity. If `refreshRow` is called after calling an updater method, but before calling the method `updateRow`, then the updates made to the row are lost. Calling the method `refreshRow` frequently will likely slow performance.

Throws:

[SQLException](#) - if a database access error occurs or if this method is called when the cursor is on the insert row

Since:

1.2

cancelRowUpdates

```
public void cancelRowUpdates()  
           throws SQLException
```

Cancels the updates made to the current row in this `ResultSet` object. This method may be called after calling an updater method(s) and before calling the method `updateRow` to roll back the updates made to a row. If no updates have been made or `updateRow` has already been called, this method has no effect.

Throws:

[SQLException](#) - if a database access error occurs or if this method is called when the cursor is on the insert row

Since:

1.2

moveToInsertRow

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

Moves the cursor to the insert row. The current cursor position is remembered while the cursor is positioned on the insert row. The insert row is a special row associated with an updatable result set. It is essentially a buffer where a new row may be constructed by calling the updater methods prior to inserting the row into the result set. Only the updater, getter, and `insertRow` methods may be called when the cursor is on the insert row. All of the columns in a result set must be given a value each time this method is called before calling `insertRow`. An updater method must be called before a getter method can be called on a column value.

Throws:

[SQLException](#) - if a database access error occurs or the result set is not updatable

Since:

1.2

moveToCurrentRow

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

Moves the cursor to the remembered cursor position, usually the current row. This method has no effect if the cursor is not on the insert row.

Throws:

[SQLException](#) - if a database access error occurs or the result set is not updatable

Since:

1.2

getStatement

```
public Statement getStatement()  
           throws SQLException
```

Retrieves the `Statement` object that produced this `ResultSet` object. If the result set was generated some other way, such as by a `DatabaseMetaData` method, this method returns `null`.

Returns:

the `Statement` object that produced this `ResultSet` object or `null` if the result set was produced some other way

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getObject

```
public Object getObject(int i,  
                        Map map)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as an `Object` in the Java programming language. If the value is an SQL `NULL`, the driver returns a Java `null`. This method uses the given `Map` object for the custom mapping of the SQL structured or distinct type that is being retrieved.

Parameters:

`i` - the first column is 1, the second is 2, ...

`map` - a `java.util.Map` object that contains the mapping from SQL type names to classes in the Java programming language

Returns:

an `Object` in the Java programming language representing the SQL value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getRef

```
public Ref getRef(int i)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `Ref` object in the Java programming language.

Parameters:

`i` - the first column is 1, the second is 2, ...

Returns:

a Ref object representing an SQL REF value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getBlob

```
public Blob getBlob(int i)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `Blob` object in the Java programming language.

Parameters:

`i` - the first column is 1, the second is 2, ...

Returns:

a `Blob` object representing the SQL BLOB value in the specified column

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getClob

```
public Clob getClob(int i)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `Clob` object in the Java programming language.

Parameters:

`i` - the first column is 1, the second is 2, ...

Returns:

a `Clob` object representing the SQL CLOB value in the specified column

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getArray

```
public Array getArray(int i)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as an `Array` object in the Java programming language.

Parameters:

`i` - the first column is 1, the second is 2, ...

Returns:

an `Array` object representing the SQL ARRAY value in the specified column

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getObject

```
public Object getObject(String colName,
    Map map)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as an `Object` in the Java programming language. If the value is an SQL NULL, the driver returns a Java `null`. This method uses the specified `Map` object for custom mapping if appropriate.

Parameters:

`colName` - the name of the column from which to retrieve the value

`map` - a `java.util.Map` object that contains the mapping from SQL type names to classes in the Java programming language

Returns:

an `Object` representing the SQL value in the specified column

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getRef

```
public Ref getRef(String colName)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `Ref` object in the Java programming language.

Parameters:

`colName` - the column name

Returns:

a `Ref` object representing the SQL REF value in the specified column

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getBlob

```
public Blob getBlob(String colName)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `Blob` object in the Java programming language.

Parameters:

`colName` - the name of the column from which to retrieve the value

Returns:

a `Blob` object representing the SQL BLOB value in the specified column

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getClob

```
public Clob getClob(String colName)  
    throws SQLException
```


Retrieves the value of the designated column in the current row of this `ResultSet` object as a `Clob` object in the Java programming language.

Parameters:

`colName` - the name of the column from which to retrieve the value

Returns:

a `Clob` object representing the SQL CLOB value in the specified column

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getArray

```
public Array getArray(String colName)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as an `Array` object in the Java programming language.

Parameters:

`colName` - the name of the column from which to retrieve the value

Returns:

an `Array` object representing the SQL ARRAY value in the specified column

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getDate

```
public Date getDate(int columnIndex,
    Calendar cal)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date` object in the Java programming language. This method uses the given calendar to construct an appropriate millisecond value for the date if the underlying database does not store timezone information.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

`cal` - the `java.util.Calendar` object to use in constructing the date

Returns:

the column value as a `java.sql.Date` object; if the value is SQL NULL, the value returned is `null` in the Java programming language

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getDate

```
public Date getDate(String columnName,  
                   Calendar cal)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date` object in the Java programming language. This method uses the given calendar to construct an appropriate millisecond value for the date if the underlying database does not store timezone information.

Parameters:

`columnName` - the SQL name of the column from which to retrieve the value

`cal` - the `java.util.Calendar` object to use in constructing the date

Returns:

the column value as a `java.sql.Date` object; if the value is SQL NULL, the value returned is `null` in the Java programming language

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getTime

```
public Time getTime(int columnIndex,  
                   Calendar cal)  
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Time` object in the Java programming language. This method uses the given calendar to construct an appropriate millisecond value for the time if the underlying database does not store timezone information.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

`cal` - the `java.util.Calendar` object to use in constructing the time

Returns:

the column value as a `java.sql.Time` object; if the value is SQL NULL, the value returned is `null` in the Java programming language

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getTime

```
public Time getTime(String columnName,
                   Calendar cal)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Time` object in the Java programming language. This method uses the given calendar to construct an appropriate millisecond value for the time if the underlying database does not store timezone information.

Parameters:

`columnName` - the SQL name of the column

`cal` - the `java.util.Calendar` object to use in constructing the time

Returns:

the column value as a `java.sql.Time` object; if the value is SQL NULL, the value returned is `null` in the Java programming language

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getTimestamp

```
public Timestamp getTimestamp(int columnIndex,
                                Calendar cal)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Timestamp` object in the Java programming language. This method uses the given calendar to construct an appropriate millisecond value for the timestamp if the underlying database does not store timezone information.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...

`cal` - the `java.util.Calendar` object to use in constructing the timestamp

Returns:

the column value as a `java.sql.Timestamp` object; if the value is SQL NULL, the value returned is `null` in the Java programming language

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getTimestamp

```
public Timestamp getTimestamp(String columnName,
                                Calendar cal)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Timestamp` object in the Java programming language. This method uses the given calendar to construct an appropriate millisecond value for the timestamp if the underlying database does not store timezone information.

Parameters:

`columnName` - the SQL name of the column

`cal` - the `java.util.Calendar` object to use in constructing the date

Returns:

the column value as a `java.sql.Timestamp` object; if the value is SQL NULL, the value returned is `null` in the Java programming language

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.2

getURL

```
public URL getURL(int columnIndex)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.net.URL` object in the Java programming language.

Parameters:

`columnIndex` - the index of the column 1 is the first, 2 is the second,...

Returns:

the column value as a `java.net.URL` object; if the value is SQL NULL, the value returned is `null` in the Java programming language

Throws:

[SQLException](#) - if a database access error occurs, or if a URL is malformed

Since:

1.4

getURL

```
public URL getURL(String columnName)
    throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.net.URL` object in the Java programming language.

Parameters:

`columnName` - the SQL name of the column

Returns:

the column value as a `java.net.URL` object; if the value is SQL NULL, the value returned is `null` in the Java programming language

Throws:

[SQLException](#) - if a database access error occurs or if a URL is malformed

Since:

1.4

updateRef

```
public void updateRef(int columnIndex,
                      Ref x)
    throws SQLException
```

Updates the designated column with a `java.sql.Ref` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.4

updateRef

```
public void updateRef(String columnName,
                      Ref x)
    throws SQLException
```

Updates the designated column with a `java.sql.Ref` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.4

updateBlob

```
public void updateBlob(int columnIndex,
                       Blob x)
```

throws [SQLException](#)

Updates the designated column with a `java.sql.Blob` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.4

updateBlob

```
public void updateBlob(String columnName,  
                        Blob x)  
    throws SQLException
```

Updates the designated column with a `java.sql.Blob` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.4

updateClob

```
public void updateClob(int columnIndex,  
                        Clob x)  
    throws SQLException
```

Updates the designated column with a `java.sql.Clob` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.4

updateClob

```
public void updateClob(String columnName,  
                        Clob x)  
    throws SQLException
```

Updates the designated column with a `java.sql.Clob` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.4

updateArray

```
public void updateArray(int columnIndex,  
                        Array x)  
    throws SQLException
```

Updates the designated column with a `java.sql.Array` value. The updater methods are

used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnIndex` - the first column is 1, the second is 2, ...
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.4

updateArray

```
public void updateArray(String columnName,
                        Array x)
    throws SQLException
```

Updates the designated column with a `java.sql.Array` value. The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database; instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters:

`columnName` - the name of the column
`x` - the new column value

Throws:

[SQLException](#) - if a database access error occurs

Since:

1.4

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | [FIELD](#) | CONSTR | [METHOD](#) DETAIL: [FIELD](#) | CONSTR | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the

[documentation redistribution policy](#).

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)*Java™ 2 Platform
Std. Ed. v1.4.2*[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#) [All Classes](#)SUMMARY: NESTED | FIELD | [CONSTR](#) | [METHOD](#) DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

java.sql

Class SQLWarning

[java.lang.Object](#)└ [java.lang.Throwable](#)└ [java.lang.Exception](#)└ [java.sql.SQLException](#)└ **java.sql.SQLWarning**

All Implemented Interfaces:

[Serializable](#)

Direct Known Subclasses:

[DataTruncation](#)

public class **SQLWarning**extends [SQLException](#)

An exception that provides information on database access warnings. Warnings are silently chained to the object whose method caused it to be reported.

Warnings may be retrieved from `Connection`, `Statement`, and `ResultSet` objects. Trying to retrieve a warning on a connection after it has been closed will cause an exception to be thrown. Similarly, trying to retrieve a warning on a statement after it has been closed or on a result set after it has been closed will cause an exception to be thrown. Note that closing a statement also closes a result set that it might have produced.

See Also:

[Connection.getWarnings\(\)](#), [Statement.getWarnings\(\)](#), [ResultSet.getWarnings\(\)](#), [Serialized Form](#)

Constructor Summary

[SQLWarning\(\)](#)

Constructs a default SQLWarning object.

[SQLWarning\(String reason\)](#)

Constructs an SQLWarning object with the given value for a reason; SQLstate defaults to null, and vendorCode defaults to 0.

[SQLWarning\(String reason, String SQLstate\)](#)

Constructs an SQLWarning object with the given reason and SQLState; the vendorCode defaults to 0.

[SQLWarning\(String reason, String SQLstate, int vendorCode\)](#)

Constructs a fully-specified SQLWarning object initialized with the given values.

Method Summary**[SQLWarning](#) [getNextWarning\(\)](#)**

Retrieves the warning chained to this SQLWarning object.

void [setNextWarning\(SQLWarning w\)](#)

Adds an SQLWarning object to the end of the chain.

Methods inherited from class java.sql.[SQLException](#)

[getErrorCode](#), [getNextException](#), [getSQLState](#), [setNextException](#)

Methods inherited from class java.lang.[Throwable](#)

[fillInStackTrace](#), [getCause](#), [getLocalizedMessage](#), [getMessage](#), [getStackTrace](#), [initCause](#), [printStackTrace](#), [printStackTrace](#), [printStackTrace](#), [setStackTrace](#), [toString](#)

Methods inherited from class java.lang.[Object](#)

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [wait](#), [wait](#), [wait](#)

Constructor Detail**SQLWarning**

```
public SQLWarning(String reason,
```

```
String SQLstate,  
int vendorCode)
```

Constructs a fully-specified `SQLWarning` object initialized with the given values.

Parameters:

`reason` - a description of the warning

`SQLstate` - an XOPEN code identifying the warning

`vendorCode` - a database vendor-specific warning code

SQLWarning

```
public SQLWarning(String reason,  
String SQLstate)
```

Constructs an `SQLWarning` object with the given reason and `SQLState`; the `vendorCode` defaults to 0.

Parameters:

`reason` - a description of the warning

`SQLstate` - an XOPEN code identifying the warning

SQLWarning

```
public SQLWarning(String reason)
```

Constructs an `SQLWarning` object with the given value for a reason; `SQLstate` defaults to `null`, and `vendorCode` defaults to 0.

Parameters:

`reason` - a description of the warning

SQLWarning

```
public SQLWarning()
```

Constructs a default `SQLWarning` object. The reason defaults to `null`, `SQLState` defaults to

`null`, and `vendorCode` defaults to 0.

Method Detail

getNextWarning

```
public SQLWarning getNextWarning()
```

Retrieves the warning chained to this `SQLWarning` object.

Returns:

the next `SQLException` in the chain; `null` if none

See Also:

[setNextWarning\(java.sql.SQLWarning\)](#)

setNextWarning

```
public void setNextWarning(SQLWarning w)
```

Adds an `SQLWarning` object to the end of the chain.

Parameters:

`w` - the new end of the `SQLException` chain

See Also:

[getNextWarning\(\)](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

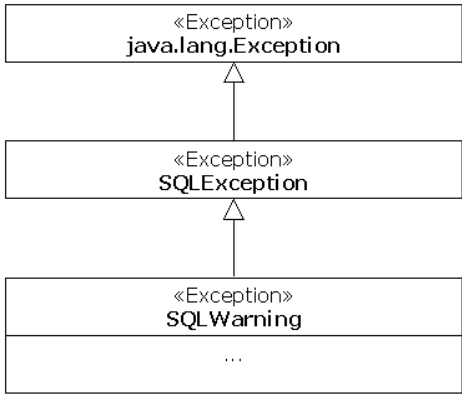
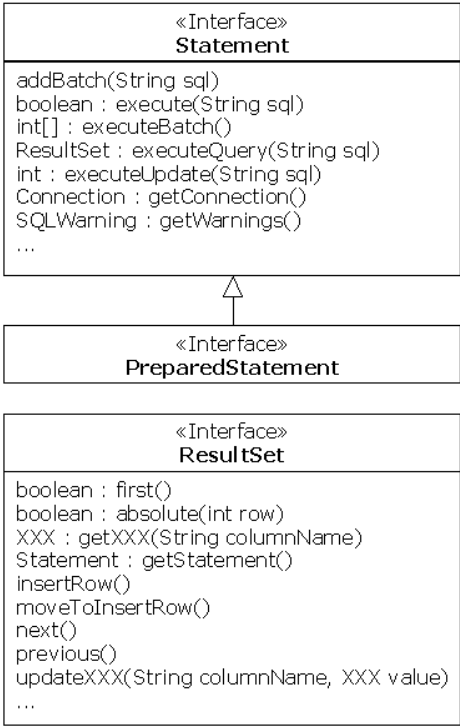
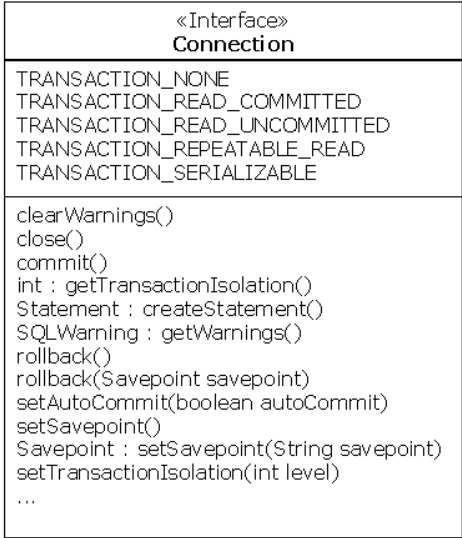
SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

java.sql



java.lang

Class Class

[java.lang.Object](#)└─ [java.lang.Class](#)**All Implemented Interfaces:**[Serializable](#)public final class **Class**extends [Object](#)implements [Serializable](#)

Instances of the class **Class** represent classes and interfaces in a running Java application. Every array also belongs to a class that is reflected as a **Class** object that is shared by all arrays with the same element type and number of dimensions. The primitive Java types (**boolean**, **byte**, **char**, **short**, **int**, **long**, **float**, and **double**), and the keyword **void** are also represented as **Class** objects.

Class has no public constructor. Instead **Class** objects are constructed automatically by the Java Virtual Machine as classes are loaded and by calls to the **defineClass** method in the class loader.

The following example uses a **Class** object to print the class name of an object:

```
void printClassName(Object obj) {
    System.out.println("The class of " + obj +
        " is " + obj.getClass().
getName());
}
```

It is also possible to get the **Class** object for a named type (or for **void**) using a class literal (JLS Section [15.8.2](#)). For example:

```
System.out.println("The name of class Foo is: "+Foo.
class.getName());
```


Since:

JDK1.0

See Also:[ClassLoader.defineClass\(byte\[\], int, int\)](#), [Serialized Form](#)**Method Summary**

<code>boolean</code>	<code>desiredAssertionStatus()</code> Returns the assertion status that would be assigned to this class if it were to be initialized at the time this method is invoked.
<code>static Class</code>	<code>forName(String className)</code> Returns the <code>Class</code> object associated with the class or interface with the given string name.
<code>static Class</code>	<code>forName(String name, boolean initialize, ClassLoader loader)</code> Returns the <code>Class</code> object associated with the class or interface with the given string name, using the given class loader.
<code>Class[]</code>	<code>getClasses()</code> Returns an array containing <code>Class</code> objects representing all the public classes and interfaces that are members of the class represented by this <code>Class</code> object.
<code>ClassLoader</code>	<code>getClassLoader()</code> Returns the class loader for the class.
<code>Class</code>	<code>getComponentType()</code> Returns the <code>Class</code> representing the component type of an array.
<code>Constructor</code>	<code>getConstructor(Class[] parameterTypes)</code> Returns a <code>Constructor</code> object that reflects the specified public constructor of the class represented by this <code>Class</code> object.
<code>Constructor[]</code>	<code>getConstructors()</code> Returns an array containing <code>Constructor</code> objects reflecting all the public constructors of the class represented by this <code>Class</code> object.
<code>Class[]</code>	<code>getDeclaredClasses()</code> Returns an array of <code>Class</code> objects reflecting all the classes and interfaces declared as members of the class represented by this <code>Class</code> object.

Constructor	getDeclaredConstructor (Class [] parameterTypes) Returns a <code>Constructor</code> object that reflects the specified constructor of the class or interface represented by this <code>Class</code> object.
Constructor []	getDeclaredConstructors () Returns an array of <code>Constructor</code> objects reflecting all the constructors declared by the class represented by this <code>Class</code> object.
Field	getDeclaredField (String name) Returns a <code>Field</code> object that reflects the specified declared field of the class or interface represented by this <code>Class</code> object.
Field []	getDeclaredFields () Returns an array of <code>Field</code> objects reflecting all the fields declared by the class or interface represented by this <code>Class</code> object.
Method	getDeclaredMethod (String name, Class [] parameterTypes) Returns a <code>Method</code> object that reflects the specified declared method of the class or interface represented by this <code>Class</code> object.
Method []	getDeclaredMethods () Returns an array of <code>Method</code> objects reflecting all the methods declared by the class or interface represented by this <code>Class</code> object.
Class	getDeclaringClass () If the class or interface represented by this <code>Class</code> object is a member of another class, returns the <code>Class</code> object representing the class in which it was declared.
Field	getField (String name) Returns a <code>Field</code> object that reflects the specified public member field of the class or interface represented by this <code>Class</code> object.
Field []	getFields () Returns an array containing <code>Field</code> objects reflecting all the accessible public fields of the class or interface represented by this <code>Class</code> object.
Class []	getInterfaces () Determines the interfaces implemented by the class or interface represented by this object.
Method	getMethod (String name, Class [] parameterTypes) Returns a <code>Method</code> object that reflects the specified public member method of the class or interface represented by this <code>Class</code> object.

Method[]	getMethods() Returns an array containing Method objects reflecting all the public <i>member</i> methods of the class or interface represented by this Class object, including those declared by the class or interface and those inherited from superclasses and superinterfaces.
int	getModifiers() Returns the Java language modifiers for this class or interface, encoded in an integer.
String	getName() Returns the name of the entity (class, interface, array class, primitive type, or void) represented by this Class object, as a String .
Package	getPackage() Gets the package for this class.
ProtectionDomain	getProtectionDomain() Returns the ProtectionDomain of this class.
URL	getResource(String name) Finds a resource with a given name.
InputStream	getResourceAsStream(String name) Finds a resource with a given name.
Object[]	getSigners() Gets the signers of this class.
Class	getSuperclass() Returns the Class representing the superclass of the entity (class, interface, primitive type or void) represented by this Class .
boolean	isArray() Determines if this Class object represents an array class.
boolean	isAssignableFrom(Class cls) Determines if the class or interface represented by this Class object is either the same as, or is a superclass or superinterface of, the class or interface represented by the specified Class parameter.
boolean	isInstance(Object obj) Determines if the specified Object is assignment-compatible with the object represented by this Class .
boolean	isInterface() Determines if the specified Class object represents an interface type.

<code>boolean</code>	<code>isPrimitive()</code> Determines if the specified <code>Class</code> object represents a primitive type.
<code>Object</code>	<code>newInstance()</code> Creates a new instance of the class represented by this <code>Class</code> object.
<code>String</code>	<code>toString()</code> Converts the object to a string.

Methods inherited from class `java.lang.Object`

[`clone`](#), [`equals`](#), [`finalize`](#), [`getClass`](#), [`hashCode`](#), [`notify`](#), [`notifyAll`](#), [`wait`](#), [`wait`](#), [`wait`](#)

Method Detail

`toString`

```
public String toString()
```

Converts the object to a string. The string representation is the string "class" or "interface", followed by a space, and then by the fully qualified name of the class in the format returned by `getName`. If this `Class` object represents a primitive type, this method returns the name of the primitive type. If this `Class` object represents void this method returns "void".

Overrides:

[`toString`](#) in class [`Object`](#)

Returns:

a string representation of this class object.

`forName`

```
public static Class forName(String className)
    throws ClassNotFoundException
```

Returns the `Class` object associated with the class or interface with the given string name. Invoking this method is equivalent to:

```
Class.forName(className, true, currentLoader)
```

where `currentLoader` denotes the defining class loader of the current class.

For example, the following code fragment returns the runtime `Class` descriptor for the class named `java.lang.Thread`:

```
Class t = Class.forName("java.lang.Thread")
```

A call to `forName("X")` causes the class named `X` to be initialized.

Parameters:

`className` - the fully qualified name of the desired class.

Returns:

the `Class` object for the class with the specified name.

Throws:

[LinkageError](#) - if the linkage fails

[ExceptionInInitializerError](#) - if the initialization provoked by this method fails

[ClassNotFoundException](#) - if the class cannot be located

forName

```
public static Class forName(String name,
                               boolean initialize,
                               ClassLoader loader)
    throws ClassNotFoundException
```

Returns the `Class` object associated with the class or interface with the given string name, using the given class loader. Given the fully qualified name for a class or interface (in the same format returned by `getName`) this method attempts to locate, load, and link the class or interface. The specified class loader is used to load the class or interface. If the parameter `loader` is null, the class is loaded through the bootstrap class loader. The class is initialized only if the `initialize` parameter is `true` and if it has not been initialized earlier.

If `name` denotes a primitive type or void, an attempt will be made to locate a user-defined class in the unnamed package whose name is `name`. Therefore, this method cannot be used to obtain any of the `Class` objects representing primitive types or void.

If `name` denotes an array class, the component type of the array class is loaded but not

initialized.

For example, in an instance method the expression:

```
Class.forName("Foo")
```

is equivalent to:

```
Class.forName("Foo", true, this.getClass().  
getClassLoader())
```

Note that this method throws errors related to loading, linking or initializing as specified in Sections 12.2, 12.3 and 12.4 of *The Java Language Specification*. Note that this method does not check whether the requested class is accessible to its caller.

If the `loader` is `null`, and a security manager is present, and the caller's class loader is not null, then this method calls the security manager's `checkPermission` method with a `RuntimePermission("getClassLoader")` permission to ensure it's ok to access the bootstrap class loader.

Parameters:

`name` - fully qualified name of the desired class

`initialize` - whether the class must be initialized

`loader` - class loader from which the class must be loaded

Returns:

class object representing the desired class

Throws:

[LinkageError](#) - if the linkage fails

[ExceptionInInitializerError](#) - if the initialization provoked by this method fails

[ClassNotFoundException](#) - if the class cannot be located by the specified class loader

Since:

1.2

See Also:

[forName\(String\)](#), [ClassLoader](#)

newInstance

```
public Object newInstance()
```

throws [InstantiationException](#),
[IllegalAccessException](#)

Creates a new instance of the class represented by this `Class` object. The class is instantiated as if by a `new` expression with an empty argument list. The class is initialized if it has not already been initialized.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.PUBLIC` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these calls could result in a `SecurityException`.

Returns:

a newly allocated instance of the class represented by this object.

Throws:

[IllegalAccessException](#) - if the class or its nullary constructor is not accessible.

[InstantiationException](#) - if this `Class` represents an abstract class, an interface, an array class, a primitive type, or void; or if the class has no nullary constructor; or if the instantiation fails for some other reason.

[ExceptionInInitializerError](#) - if the initialization provoked by this method fails.

[SecurityException](#) - if there is no permission to create a new instance.

isInstance

```
public boolean isInstance(Object obj)
```

Determines if the specified `Object` is assignment-compatible with the object represented by this `Class`. This method is the dynamic equivalent of the Java language `instanceof` operator. The method returns `true` if the specified `Object` argument is non-null and can be cast to the reference type represented by this `Class` object without raising a `ClassCastException`. It returns `false` otherwise.

Specifically, if this `Class` object represents a declared class, this method returns `true` if the specified `Object` argument is an instance of the represented class (or of any of its subclasses); it returns `false` otherwise. If this `Class` object represents an array class, this method returns `true` if the specified `Object` argument can be converted to an object of the array class by an identity conversion or by a widening reference conversion; it returns `false` otherwise. If this `Class` object represents an interface, this method returns `true` if the class or any superclass of the specified `Object` argument implements this interface; it returns

`false` otherwise. If this `Class` object represents a primitive type, this method returns `false`.

Parameters:

`obj` - the object to check

Returns:

true if `obj` is an instance of this class

Since:

JDK1.1

isAssignableFrom

```
public boolean isAssignableFrom(Class cls)
```

Determines if the class or interface represented by this `Class` object is either the same as, or is a superclass or superinterface of, the class or interface represented by the specified `Class` parameter. It returns `true` if so; otherwise it returns `false`. If this `Class` object represents a primitive type, this method returns `true` if the specified `Class` parameter is exactly this `Class` object; otherwise it returns `false`.

Specifically, this method tests whether the type represented by the specified `Class` parameter can be converted to the type represented by this `Class` object via an identity conversion or via a widening reference conversion. See *The Java Language Specification*, sections 5.1.1 and 5.1.4, for details.

Parameters:

`cls` - the `Class` object to be checked

Returns:

the `boolean` value indicating whether objects of the type `cls` can be assigned to objects of this class

Throws:

[NullPointerException](#) - if the specified `Class` parameter is null.

Since:

JDK1.1

isInterface

```
public boolean isInterface()
```

Determines if the specified `Class` object represents an interface type.

Returns:

`true` if this object represents an interface; `false` otherwise.

isArray

```
public boolean isArray()
```

Determines if this `Class` object represents an array class.

Returns:

`true` if this object represents an array class; `false` otherwise.

Since:

JDK1.1

isPrimitive

```
public boolean isPrimitive()
```

Determines if the specified `Class` object represents a primitive type.

There are nine predefined `Class` objects to represent the eight primitive types and void. These are created by the Java Virtual Machine, and have the same names as the primitive types that they represent, namely `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`.

These objects may only be accessed via the following public static final variables, and are the only `Class` objects for which this method returns `true`.

Returns:

`true` if and only if this class represents a primitive type

Since:

JDK1.1

See Also:

[`Boolean.TYPE`](#), [`Character.TYPE`](#), [`Byte.TYPE`](#), [`Short.TYPE`](#), [`Integer.TYPE`](#), [`Long.TYPE`](#), [`Float.TYPE`](#), [`Double.TYPE`](#), [`Void.TYPE`](#)

getName

```
public String getName()
```

Returns the name of the entity (class, interface, array class, primitive type, or void) represented by this `Class` object, as a `String`.

If this class object represents a reference type that is not an array type then the binary name of the class is returned, as specified by the Java Language Specification, Second Edition.

If this class object represents a primitive type or void, then the name returned is a `String` equal to the Java language keyword corresponding to the primitive type or void.

If this class object represents a class of arrays, then the internal form of the name consists of the name of the element type preceded by one or more '[' characters representing the depth of the array nesting. The encoding of element type names is as follows:

Element Type	Encoding
boolean	Z
byte	B
char	C
class or interface	<i>Lclassname;</i>
double	D
float	F
int	I
long	J
short	S

The class or interface name *classname* is the binary name of the class specified above.

Examples:

```
String.class.getName()
    returns "java.lang.String"
byte.class.getName()
    returns "byte"
(new Object[3]).getClass().getName()
    returns "[Ljava.lang.Object;"
(new int[3][4][5][6][7][8][9]).getClass().getName()
    returns "[[[[[[[I"
```

Returns:

the name of the class or interface represented by this object.

getClassLoader

```
public ClassLoader getClassLoader()
```

Returns the class loader for the class. Some implementations may use null to represent the bootstrap class loader. This method will return null in such implementations if this class was loaded by the bootstrap class loader.

If a security manager is present, and the caller's class loader is not null and the caller's class loader is not the same as or an ancestor of the class loader for the class whose class loader is requested, then this method calls the security manager's `checkPermission` method with a `RuntimePermission("getClassLoader")` permission to ensure it's ok to access the class loader for the class.

If this object represents a primitive type or void, null is returned.

Returns:

the class loader that loaded the class or interface represented by this object.

Throws:

[SecurityException](#) - if a security manager exists and its `checkPermission` method denies access to the class loader for the class.

See Also:

[ClassLoader](#), [SecurityManager.checkPermission\(java.security.Permission\)](#), [RuntimePermission](#)

getSuperclass

```
public Class getSuperclass()
```

Returns the `Class` representing the superclass of the entity (class, interface, primitive type or void) represented by this `Class`. If this `Class` represents either the `Object` class, an interface, a primitive type, or void, then null is returned. If this object represents an array class then the `Class` object representing the `Object` class is returned.

Returns:

the superclass of the class represented by this object.

getPackage

public [Package](#) **getPackage()**

Gets the package for this class. The class loader of this class is used to find the package. If the class was loaded by the bootstrap class loader the set of packages loaded from CLASSPATH is searched to find the package of the class. Null is returned if no package object was created by the class loader of this class.

Packages have attributes for versions and specifications only if the information was defined in the manifests that accompany the classes, and if the class loader created the package instance with the attributes from the manifest.

Returns:

the package of the class, or null if no package information is available from the archive or codebase.

getInterfaces

public [Class](#)[] **getInterfaces()**

Determines the interfaces implemented by the class or interface represented by this object.

If this object represents a class, the return value is an array containing objects representing all interfaces implemented by the class. The order of the interface objects in the array corresponds to the order of the interface names in the `implements` clause of the declaration of the class represented by this object. For example, given the declaration:

```
class Shimmer implements FloorWax, DessertTopping
{ ... }
```

suppose the value of `s` is an instance of `Shimmer`; the value of the expression:

```
s.getClass().getInterfaces()[0]
```

is the `Class` object that represents interface `FloorWax`; and the value of:

```
s.getClass().getInterfaces()[1]
```

is the `Class` object that represents interface `DessertTopping`.

If this object represents an interface, the array contains objects representing all interfaces extended by the interface. The order of the interface objects in the array corresponds to the order of the interface names in the `extends` clause of the declaration of the interface represented by this object.

If this object represents a class or interface that implements no interfaces, the method returns an array of length 0.

If this object represents a primitive type or void, the method returns an array of length 0.

Returns:

an array of interfaces implemented by this class.

getComponentType

```
public Class getComponentType()
```

Returns the `Class` representing the component type of an array. If this class does not represent an array class this method returns null.

Returns:

the `Class` representing the component type of this class if this class is an array

Since:

JDK1.1

See Also:

[Array](#)

getModifiers

```
public int getModifiers()
```

Returns the Java language modifiers for this class or interface, encoded in an integer. The modifiers consist of the Java Virtual Machine's constants for `public`, `protected`, `private`, `final`, `static`, `abstract` and `interface`; they should be decoded using the methods of class `Modifier`.

If the underlying class is an array class, then its `public`, `private` and `protected` modifiers are the same as those of its component type. If this `Class` represents a primitive

type or void, its `public` modifier is always `true`, and its `protected` and `private` modifiers are always `false`. If this object represents an array class, a primitive type or void, then its `final` modifier is always `true` and its interface modifier is always `false`. The values of its other modifiers are not determined by this specification.

The modifier encodings are defined in *The Java Virtual Machine Specification*, table 4.1.

Returns:

the `int` representing the modifiers for this class

Since:

JDK1.1

See Also:

[Modifier](#)

getSigners

```
public Object[] getSigners()
```

Gets the signers of this class.

Returns:

the signers of this class, or null if there are no signers. In particular, this method returns null if this object represents a primitive type or void.

Since:

JDK1.1

getDeclaringClass

```
public Class getDeclaringClass()
```

If the class or interface represented by this `Class` object is a member of another class, returns the `Class` object representing the class in which it was declared. This method returns null if this class or interface is not a member of any other class. If this `Class` object represents an array class, a primitive type, or void, then this method returns null.

Returns:

the declaring class for this class

Since:

JDK1.1

getClasses

```
public Class[] getClasses()
```

Returns an array containing `Class` objects representing all the public classes and interfaces that are members of the class represented by this `Class` object. This includes public class and interface members inherited from superclasses and public class and interface members declared by the class. This method returns an array of length 0 if this `Class` object has no public member classes or interfaces. This method also returns an array of length 0 if this `Class` object represents a primitive type, an array class, or void.

For this class and each of its superclasses, the following security checks are performed: If there is a security manager, the security manager's `checkMemberAccess` method is called with `this` and `Member.PUBLIC` as its arguments, where `this` is this class or the superclass whose members are being determined. If the class is in a package, then the security manager's `checkPackageAccess` method is also called with the package name as its argument. Either of these calls could result in a `SecurityException`.

Returns:

the array of `Class` objects representing the public members of this class

Throws:

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[SecurityManager.checkMemberAccess\(Class, int\)](#),
[SecurityManager.checkPackageAccess\(String\)](#)

getFields

```
public Field[] getFields()  
    throws SecurityException
```

Returns an array containing `Field` objects reflecting all the accessible public fields of the class or interface represented by this `Class` object. The elements in the array returned are not sorted and are not in any particular order. This method returns an array of length 0 if the class or interface has no accessible public fields, or if it represents an array class, a primitive type, or void.

Specifically, if this `Class` object represents a class, this method returns the public fields of

this class and of all its superclasses. If this `Class` object represents an interface, this method returns the fields of this interface and of all its superinterfaces.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.PUBLIC` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these calls could result in a `SecurityException`.

The implicit length field for array class is not reflected by this method. User code should use the methods of class `Array` to manipulate arrays.

See *The Java Language Specification*, sections 8.2 and 8.3.

Returns:

the array of `Field` objects representing the public fields

Throws:

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[Field](#), [SecurityManager.checkMemberAccess\(Class, int\)](#),
[SecurityManager.checkPackageAccess\(String\)](#)

getMethods

```
public Method[] getMethods()
           throws SecurityException
```

Returns an array containing `Method` objects reflecting all the public *member* methods of the class or interface represented by this `Class` object, including those declared by the class or interface and those inherited from superclasses and superinterfaces. The elements in the array returned are not sorted and are not in any particular order. This method returns an array of length 0 if this `Class` object represents a class or interface that has no public member methods, or if this `Class` object represents an array class, primitive type, or void.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.PUBLIC` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these calls could result in a `SecurityException`.

The class initialization method `<clinit>` is not included in the returned array. If the class declares multiple public member methods with the same parameter types, they are all included in the returned array.

See *The Java Language Specification*, sections 8.2 and 8.4.

Returns:

the array of `Method` objects representing the public methods of this class

Throws:

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[Method](#), [SecurityManager.checkMemberAccess\(Class, int\)](#),
[SecurityManager.checkPackageAccess\(String\)](#)

getConstructors

```
public Constructor[] getConstructors()
                        throws SecurityException
```

Returns an array containing `Constructor` objects reflecting all the public constructors of the class represented by this `Class` object. An array of length 0 is returned if the class has no public constructors, or if the class is an array class, or if the class reflects a primitive type or void.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.PUBLIC` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these calls could result in a `SecurityException`.

Returns:

the array containing `Method` objects for all the declared public constructors of this class matches the specified `parameterTypes`

Throws:

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[Constructor](#), [SecurityManager.checkMemberAccess\(Class, int\)](#),
[SecurityManager.checkPackageAccess\(String\)](#)

getField

```
public Field getField(String name)
    throws NoSuchFieldException,
           SecurityException
```

Returns a `Field` object that reflects the specified public member field of the class or interface represented by this `Class` object. The `name` parameter is a `String` specifying the simple name of the desired field.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.PUBLIC` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these calls could result in a `SecurityException`.

The field to be reflected is determined by the algorithm that follows. Let `C` be the class represented by this object:

1. If `C` declares a public field with the name specified, that is the field to be reflected.
2. If no field was found in step 1 above, this algorithm is applied recursively to each direct superinterface of `C`. The direct superinterfaces are searched in the order they were declared.
3. If no field was found in steps 1 and 2 above, and `C` has a superclass `S`, then this algorithm is invoked recursively upon `S`. If `C` has no superclass, then a `NoSuchFieldException` is thrown.

See *The Java Language Specification*, sections 8.2 and 8.3.

Parameters:

`name` - the field name

Returns:

the `Field` object of this class specified by `name`

Throws:

[NoSuchFieldException](#) - if a field with the specified name is not found.

[NullPointerException](#) - if `name` is `null`

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[Field](#), [SecurityManager.checkMemberAccess\(Class, int\)](#),

[SecurityManager.checkPackageAccess\(String\)](#)

getMethod

```
public Method getMethod(String name,  
                        Class[] parameterTypes)  
    throws NoSuchMethodException,  
           SecurityException
```

Returns a `Method` object that reflects the specified public member method of the class or interface represented by this `Class` object. The `name` parameter is a `String` specifying the simple name the desired method. The `parameterTypes` parameter is an array of `Class` objects that identify the method's formal parameter types, in declared order. If `parameterTypes` is `null`, it is treated as if it were an empty array.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.PUBLIC` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these calls could result in a `SecurityException`.

If the `name` is "`<init>`" or "`<clinit>`" a `NoSuchMethodException` is raised. Otherwise, the method to be reflected is determined by the algorithm that follows. Let `C` be the class represented by this object:

1. `C` is searched for any *matching methods*. If no matching method is found, the algorithm of step 1 is invoked recursively on the superclass of `C`.
2. If no method was found in step 1 above, the superinterfaces of `C` are searched for a matching method. If any such method is found, it is reflected.

To find a matching method in a class `C`: If `C` declares exactly one public method with the specified name and exactly the same formal parameter types, that is the method reflected. If more than one such method is found in `C`, and one of these methods has a return type that is more specific than any of the others, that method is reflected; otherwise one of the methods is chosen arbitrarily.

See *The Java Language Specification*, sections 8.2 and 8.4.

Parameters:

`name` - the name of the method

`parameterTypes` - the list of parameters

Returns:

the `Method` object that matches the specified name and `parameterTypes`

Throws:

[NoSuchMethodException](#) - if a matching method is not found or if the name is "<init>" or "<clinit>".

[NullPointerException](#) - if name is null

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[Method](#), [SecurityManager.checkMemberAccess\(Class, int\)](#),
[SecurityManager.checkPackageAccess\(String\)](#)

getConstructor

```
public Constructor getConstructor(Class[] parameterTypes)
                                   throws NoSuchMethodException,
                                           SecurityException
```

Returns a `Constructor` object that reflects the specified public constructor of the class represented by this `Class` object. The `parameterTypes` parameter is an array of `Class` objects that identify the constructor's formal parameter types, in declared order.

The constructor to reflect is the public constructor of the class represented by this `Class` object whose formal parameter types match those specified by `parameterTypes`.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.PUBLIC` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these calls could result in a `SecurityException`.

Parameters:

`parameterTypes` - the parameter array

Returns:

the `Method` object of the public constructor that matches the specified `parameterTypes`

Throws:

[NoSuchMethodException](#) - if a matching method is not found.

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[Constructor](#), [SecurityManager.checkMemberAccess\(Class, int\)](#),

[SecurityManager.checkPackageAccess\(String\)](#)

getDeclaredClasses

```
public Class[] getDeclaredClasses()  
           throws SecurityException
```

Returns an array of `Class` objects reflecting all the classes and interfaces declared as members of the class represented by this `Class` object. This includes public, protected, default (package) access, and private classes and interfaces declared by the class, but excludes inherited classes and interfaces. This method returns an array of length 0 if the class declares no classes or interfaces as members, or if this `Class` object represents a primitive type, an array class, or void.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.DECLARED` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these calls could result in a `SecurityException`.

Returns:

the array of `Class` objects representing all the declared members of this class

Throws:

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[SecurityManager.checkMemberAccess\(Class, int\)](#),
[SecurityManager.checkPackageAccess\(String\)](#)

getDeclaredFields

```
public Field[] getDeclaredFields()  
           throws SecurityException
```

Returns an array of `Field` objects reflecting all the fields declared by the class or interface represented by this `Class` object. This includes public, protected, default (package) access, and private fields, but excludes inherited fields. The elements in the array returned are not sorted and are not in any particular order. This method returns an array of length 0 if the class or interface declares no fields, or if this `Class` object represents a primitive type, an array

class, or void.

See *The Java Language Specification*, sections 8.2 and 8.3.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.DECLARED` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these calls could result in a `SecurityException`.

Returns:

the array of `Field` objects representing all the declared fields of this class

Throws:

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[Field](#), [SecurityManager.checkMemberAccess\(Class, int\)](#),
[SecurityManager.checkPackageAccess\(String\)](#)

getDeclaredMethods

```
public Method[] getDeclaredMethods()
           throws SecurityException
```

Returns an array of `Method` objects reflecting all the methods declared by the class or interface represented by this `Class` object. This includes public, protected, default (package) access, and private methods, but excludes inherited methods. The elements in the array returned are not sorted and are not in any particular order. This method returns an array of length 0 if the class or interface declares no methods, or if this `Class` object represents a primitive type, an array class, or void. The class initialization method `<clinit>` is not included in the returned array. If the class declares multiple public member methods with the same parameter types, they are all included in the returned array.

See *The Java Language Specification*, section 8.2.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.DECLARED` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these calls could result in a `SecurityException`.

Returns:

the array of `Method` objects representing all the declared methods of this class

Throws:

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[Method](#), [SecurityManager.checkMemberAccess\(Class, int\)](#),
[SecurityManager.checkPackageAccess\(String\)](#)

getDeclaredConstructors

```
public Constructor[] getDeclaredConstructors()
                               throws SecurityException
```

Returns an array of `Constructor` objects reflecting all the constructors declared by the class represented by this `Class` object. These are public, protected, default (package) access, and private constructors. The elements in the array returned are not sorted and are not in any particular order. If the class has a default constructor, it is included in the returned array. This method returns an array of length 0 if this `Class` object represents an interface, a primitive type, an array class, or void.

See *The Java Language Specification*, section 8.2.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.DECLARED` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these calls could result in a `SecurityException`.

Returns:

the array of `Method` objects representing all the declared constructors of this class

Throws:

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[Constructor](#), [SecurityManager.checkMemberAccess\(Class, int\)](#),
[SecurityManager.checkPackageAccess\(String\)](#)

getDeclaredField

```
public Field getDeclaredField(String name)
                               throws NoSuchFieldException,
                                       SecurityException
```

Returns a `Field` object that reflects the specified declared field of the class or interface represented by this `Class` object. The `name` parameter is a `String` that specifies the simple name of the desired field. Note that this method will not reflect the `length` field of an array class.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.DECLARED` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these calls could result in a `SecurityException`.

Parameters:

`name` - the name of the field

Returns:

the `Field` object for the specified field in this class

Throws:

[NoSuchFieldException](#) - if a field with the specified name is not found.

[NullPointerException](#) - if `name` is `null`

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[Field](#), [SecurityManager.checkMemberAccess\(Class, int\)](#),
[SecurityManager.checkPackageAccess\(String\)](#)

getDeclaredMethod

```
public Method getDeclaredMethod(String name,
                                Class[] parameterTypes)
                               throws NoSuchMethodException,
                                       SecurityException
```

Returns a `Method` object that reflects the specified declared method of the class or interface represented by this `Class` object. The `name` parameter is a `String` that specifies the simple name of the desired method, and the `parameterTypes` parameter is an array of `Class`

objects that identify the method's formal parameter types, in declared order. If more than one method with the same parameter types is declared in a class, and one of these methods has a return type that is more specific than any of the others, that method is returned; otherwise one of the methods is chosen arbitrarily. If the name is "<init>" or "<clinit>" a `NoSuchMethodException` is raised.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.DECLARED` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these calls could result in a `SecurityException`.

Parameters:

`name` - the name of the method

`parameterTypes` - the parameter array

Returns:

the `Method` object for the method of this class matching the specified name and parameters

Throws:

[NoSuchMethodException](#) - if a matching method is not found.

[NullPointerException](#) - if `name` is `null`

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[Method](#), [SecurityManager.checkMemberAccess\(Class, int\)](#),
[SecurityManager.checkPackageAccess\(String\)](#)

getDeclaredConstructor

```
public Constructor getDeclaredConstructor(Class[] parameterTypes)
                                   throws NoSuchMethodException,
                                           SecurityException
```

Returns a `Constructor` object that reflects the specified constructor of the class or interface represented by this `Class` object. The `parameterTypes` parameter is an array of `Class` objects that identify the constructor's formal parameter types, in declared order.

If there is a security manager, this method first calls the security manager's `checkMemberAccess` method with `this` and `Member.DECLARED` as its arguments. If the class is in a package, then this method also calls the security manager's `checkPackageAccess` method with the package name as its argument. Either of these

calls could result in a `SecurityException`.

Parameters:

`parameterTypes` - the parameter array

Returns:

The `Method` object for the constructor with the specified parameter list

Throws:

[NoSuchMethodException](#) - if a matching method is not found.

[SecurityException](#) - if access to the information is denied.

Since:

JDK1.1

See Also:

[Constructor](#), [SecurityManager.checkMemberAccess\(Class, int\)](#),
[SecurityManager.checkPackageAccess\(String\)](#)

getResourceAsStream

```
public InputStream getResourceAsStream(String name)
```

Finds a resource with a given name. This method returns null if no resource with this name is found. The rules for searching resources associated with a given class are implemented by the defining class loader of the class.

This method delegates the call to its class loader, after making these changes to the resource name: if the resource name starts with "/", it is unchanged; otherwise, the package name is prepended to the resource name after converting "." to "/". If this object was loaded by the bootstrap loader, the call is delegated to `ClassLoader`.
`getSystemResourceAsStream`.

Parameters:

`name` - name of the desired resource

Returns:

a `java.io.InputStream` object.

Throws:

[NullPointerException](#) - if `name` is null.

Since:

JDK1.1

See Also:

[ClassLoader](#)

getResource

```
public URL getResource(String name)
```

Finds a resource with a given name. This method returns null if no resource with this name is found. The rules for searching resources associated with a given class are implemented by the * defining class loader of the class.

This method delegates the call to its class loader, after making these changes to the resource name: if the resource name starts with "/", it is unchanged; otherwise, the package name is prepended to the resource name after converting "." to "/". If this object was loaded by the bootstrap loader, the call is delegated to `ClassLoader.getResource`.

Parameters:

name - name of the desired resource

Returns:

a `java.net.URL` object.

Since:

JDK1.1

See Also:

[ClassLoader](#)

getProtectionDomain

```
public ProtectionDomain getProtectionDomain()
```

Returns the `ProtectionDomain` of this class. If there is a security manager installed, this method first calls the security manager's `checkPermission` method with a `RuntimePermission("getProtectionDomain")` permission to ensure it's ok to get the `ProtectionDomain`.

Returns:

the `ProtectionDomain` of this class

Throws:

[SecurityException](#) - if a security manager exists and its `checkPermission` method doesn't allow getting the `ProtectionDomain`.

Since:

1.2

See Also:

[ProtectionDomain](#), [SecurityManager.checkPermission\(java.security.Permission\)](#), [RuntimePermission](#)

desiredAssertionStatus

```
public boolean desiredAssertionStatus()
```

Returns the assertion status that would be assigned to this class if it were to be initialized at the time this method is invoked. If this class has had its assertion status set, the most recent setting will be returned; otherwise, if any package default assertion status pertains to this class, the most recent setting for the most specific pertinent package default assertion status is returned; otherwise, if this class is not a system class (i.e., it has a class loader) its class loader's default assertion status is returned; otherwise, the system class default assertion status is returned.

Few programmers will have any need for this method; it is provided for the benefit of the JRE itself. (It allows a class to determine at the time that it is initialized whether assertions should be enabled.) Note that this method is not guaranteed to return the actual assertion status that was (or will be) associated with the specified class when it was (or will be) initialized.

Returns:

the desired assertion status of the specified class.

Since:

1.4

See Also:

[ClassLoader.setClassAssertionStatus\(java.lang.String, boolean\)](#), [ClassLoader.setPackageAssertionStatus\(java.lang.String, boolean\)](#), [ClassLoader.setDefaultAssertionStatus\(boolean\)](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | FIELD | CONSTR | [METHOD](#) DETAIL: FIELD | CONSTR | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

java.sql

Interface Driver

public interface **Driver**

The interface that every driver class must implement.

The Java SQL framework allows for multiple database drivers.

Each driver should supply a class that implements the Driver interface.

The DriverManager will try to load as many drivers as it can find and then for any given connection request, it will ask each driver in turn to try to connect to the target URL.

It is strongly recommended that each Driver class should be small and standalone so that the Driver class can be loaded and queried without bringing in vast quantities of supporting code.

When a Driver class is loaded, it should create an instance of itself and register it with the DriverManager. This means that a user can load and register a driver by calling

```
Class.forName("foo.bah.Driver")
```

See Also:

[DriverManager](#), [Connection](#)

Method Summary

boolean	acceptsURL (String url)
	Retrieves whether the driver thinks that it can open a connection to the given URL.
Connection	connect (String url, Properties info)
	Attempts to make a database connection to the given URL.

int	<code>getMajorVersion()</code> Retrieves the driver's major version number.
int	<code>getMinorVersion()</code> Gets the driver's minor version number.
<code>DriverPropertyInfo</code> []	<code>getPropertyInfo(String url, Properties info)</code> Gets information about the possible properties for this driver.
boolean	<code>jdbcCompliant()</code> Reports whether this driver is a genuine JDBC Compliant™ driver.

Method Detail

connect

```
public Connection connect(String url,
                           Properties info)
    throws SQLException
```

Attempts to make a database connection to the given URL. The driver should return "null" if it realizes it is the wrong kind of driver to connect to the given URL. This will be common, as when the JDBC driver manager is asked to connect to a given URL it passes the URL to each loaded driver in turn.

The driver should throw an `SQLException` if it is the right driver to connect to the given URL but has trouble connecting to the database.

The `java.util.Properties` argument can be used to pass arbitrary string tag/value pairs as connection arguments. Normally at least "user" and "password" properties should be included in the `Properties` object.

Parameters:

`url` - the URL of the database to which to connect

`info` - a list of arbitrary string tag/value pairs as connection arguments. Normally at least a "user" and "password" property should be included.

Returns:

a `Connection` object that represents a connection to the URL

Throws:

[SQLException](#) - if a database access error occurs

acceptsURL

```
public boolean acceptsURL(String url)
                        throws SQLException
```

Retrieves whether the driver thinks that it can open a connection to the given URL. Typically drivers will return `true` if they understand the subprotocol specified in the URL and `false` if they do not.

Parameters:

`url` - the URL of the database

Returns:

`true` if this driver understands the given URL; `false` otherwise

Throws:

[SQLException](#) - if a database access error occurs

getPropertyInfo

```
public DriverPropertyInfo[] getPropertyInfo(String url,
                                             Properties info)
                        throws SQLException
```

Gets information about the possible properties for this driver.

The `getPropertyInfo` method is intended to allow a generic GUI tool to discover what properties it should prompt a human for in order to get enough information to connect to a database. Note that depending on the values the human has supplied so far, additional values may become necessary, so it may be necessary to iterate through several calls to the `getPropertyInfo` method.

Parameters:

`url` - the URL of the database to which to connect

`info` - a proposed list of tag/value pairs that will be sent on connect open

Returns:

an array of `DriverPropertyInfo` objects describing possible properties. This array may be an empty array if no properties are required.

Throws:

[SQLException](#) - if a database access error occurs

getMajorVersion

```
public int getMajorVersion()
```

Retrieves the driver's major version number. Initially this should be 1.

Returns:

this driver's major version number

getMinorVersion

```
public int getMinorVersion()
```

Gets the driver's minor version number. Initially this should be 0.

Returns:

this driver's minor version number

jdbcCompliant

```
public boolean jdbcCompliant()
```

Reports whether this driver is a genuine JDBC CompliantTM driver. A driver may only report `true` here if it passes the JDBC compliance tests; otherwise it is required to return `false`.

JDBC compliance requires full support for the JDBC API and full support for SQL 92 Entry Level. It is expected that JDBC compliant drivers will be available for all the major commercial databases.

This method is not intended to encourage the development of non-JDBC compliant drivers, but is a recognition of the fact that some vendors are interested in using the JDBC API and framework for lightweight databases that do not support full database functionality, or for special databases such as document information retrieval where a SQL implementation may not be feasible.

Returns:

`true` if this driver is JDBC Compliant; `false` otherwise

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)*Java™ 2 Platform
Std. Ed. v1.4.2*[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#) [All Classes](#)SUMMARY: NESTED | FIELD | CONSTR | [METHOD](#) DETAIL: FIELD | CONSTR | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

java.sql

Class DriverManager

[java.lang.Object](#)└─ [java.sql.DriverManager](#)public class **DriverManager**extends [Object](#)

The basic service for managing a set of JDBC drivers.

NOTE: The `DataSource` interface, new in the JDBC 2.0 API, provides another way to connect to a data source. The use of a `DataSource` object is the preferred means of connecting to a data source.

As part of its initialization, the `DriverManager` class will attempt to load the driver classes referenced in the "jdbc.drivers" system property. This allows a user to customize the JDBC Drivers used by their applications. For example in your `~/.hotjava/properties` file you might specify:

```
jdbc.drivers=foo.bah.Driver:wombat.sql.Driver:bad.taste.ourDriver
```

A program can also explicitly load JDBC drivers at any time. For example, the `my.sql.Driver` is loaded with the following statement:

```
Class.forName("my.sql.Driver");
```

When the method `getConnection` is called, the `DriverManager` will attempt to locate a suitable driver from amongst those loaded at initialization and those loaded explicitly using the same classloader as the current applet or application.

Starting with the Java 2 SDK, Standard Edition, version 1.3, a logging stream can be set only if the proper permission has been granted. Normally this will be done with the tool `PolicyTool`, which can be used to grant permission `java.sql.SQLPermission "setLog"`.

See Also:

[Driver, Connection](#)**Method Summary**

static void	deregisterDriver (Driver driver) Drops a driver from the DriverManager's list.
static Connection	getConnection (String url) Attempts to establish a connection to the given database URL.
static Connection	getConnection (String url, Properties info) Attempts to establish a connection to the given database URL.
static Connection	getConnection (String url, String user, String password) Attempts to establish a connection to the given database URL.
static Driver	getDriver (String url) Attempts to locate a driver that understands the given URL.
static Enumeration	getDrivers () Retrieves an Enumeration with all of the currently loaded JDBC drivers to which the current caller has access.
static int	getLoginTimeout () Gets the maximum time in seconds that a driver can wait when attempting to log in to a database.
static PrintStream	getLogStream () Deprecated.
static PrintWriter	getLogWriter () Retrieves the log writer.
static void	println (String message) Prints a message to the current JDBC log stream.
static void	registerDriver (Driver driver) Registers the given driver with the DriverManager.
static void	setLoginTimeout (int seconds) Sets the maximum time in seconds that a driver will wait while attempting to connect to a database.
static void	setLogStream (PrintStream out) Deprecated.

static void	setLogWriter (PrintWriter out)

Sets the logging/tracing <code>PrintWriter</code> object that is used by the <code>DriverManager</code> and all drivers.

Methods inherited from class `java.lang.Object`

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

Method Detail

getLogWriter

```
public static PrintWriter getLogWriter()
```

Retrieves the log writer. The `getLogWriter` and `setLogWriter` methods should be used instead of the `get/set logStream` methods, which are deprecated.

Returns:

a `java.io.PrintWriter` object

Since:

1.2

See Also:

[setLogWriter\(java.io.PrintWriter\)](#)

setLogWriter

```
public static void setLogWriter(PrintWriter out)
```

Sets the logging/tracing `PrintWriter` object that is used by the `DriverManager` and all drivers.

There is a minor versioning problem created by the introduction of the method `setLogWriter`. The method `setLogWriter` cannot create a `PrintStream` object that will be returned by `getLogStream`---the Java platform does not provide a backward conversion. As a result, a new application that uses `setLogWriter` and also uses a JDBC 1.0 driver that uses `getLogStream` will likely not see debugging information written by that driver.

In the Java 2 SDK, Standard Edition, version 1.3 release, this method checks to see that there is an `SQLPermission` object before setting the logging stream. If a `SecurityManager` exists and its `checkPermission` method denies setting the log writer, this method throws a `java.lang.SecurityException`.

Parameters:

`out` - the new logging/tracing `PrintStream` object; `null` to disable logging and tracing

Throws:

[SecurityException](#) - if a security manager exists and its `checkPermission` method denies setting the log writer

Since:

1.2

See Also:

[SecurityManager.checkPermission\(java.security.Permission\),
getLogWriter\(\)](#)

getConnection

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

Attempts to establish a connection to the given database URL. The `DriverManager` attempts to select an appropriate driver from the set of registered JDBC drivers.

Parameters:

`url` - a database url of the form `jdbc:subprotocol:subname`

`info` - a list of arbitrary string tag/value pairs as connection arguments; normally at least a "user" and "password" property should be included

Returns:

a `Connection` to the URL

Throws:

[SQLException](#) - if a database access error occurs

getConnection

```
public static Connection getConnection(String url,
                                         String user,
```

[String](#) password)
throws [SQLException](#)

Attempts to establish a connection to the given database URL. The `DriverManager` attempts to select an appropriate driver from the set of registered JDBC drivers.

Parameters:

`url` - a database url of the form `jdbc:subprotocol:subname`
`user` - the database user on whose behalf the connection is being made
`password` - the user's password

Returns:

a connection to the URL

Throws:

[SQLException](#) - if a database access error occurs

getConnection

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

Attempts to establish a connection to the given database URL. The `DriverManager` attempts to select an appropriate driver from the set of registered JDBC drivers.

Parameters:

`url` - a database url of the form `jdbc:subprotocol:subname`

Returns:

a connection to the URL

Throws:

[SQLException](#) - if a database access error occurs

getDriver

```
public static Driver getDriver(String url)  
                                   throws SQLException
```

Attempts to locate a driver that understands the given URL. The `DriverManager` attempts to select an appropriate driver from the set of registered JDBC drivers.

Parameters:

`url` - a database URL of the form `jdbc:subprotocol:subname`

Returns:

a `Driver` object representing a driver that can connect to the given URL

Throws:

[SQLException](#) - if a database access error occurs

registerDriver

```
public static void registerDriver(Driver driver)
                               throws SQLException
```

Registers the given driver with the `DriverManager`. A newly-loaded driver class should call the method `registerDriver` to make itself known to the `DriverManager`.

Parameters:

`driver` - the new JDBC Driver that is to be registered with the `DriverManager`

Throws:

[SQLException](#) - if a database access error occurs

deregisterDriver

```
public static void deregisterDriver(Driver driver)
                                   throws SQLException
```

Drops a driver from the `DriverManager`'s list. Applets can only deregister drivers from their own classloaders.

Parameters:

`driver` - the JDBC Driver to drop

Throws:

[SQLException](#) - if a database access error occurs

getDrivers

```
public static Enumeration getDrivers()
```

Retrieves an `Enumeration` with all of the currently loaded JDBC drivers to which the current

caller has access.

Note: The classname of a driver can be found using `d.getClass().getName()`

Returns:

the list of JDBC Drivers loaded by the caller's class loader

setLoginTimeout

```
public static void setLoginTimeout(int seconds)
```

Sets the maximum time in seconds that a driver will wait while attempting to connect to a database.

Parameters:

seconds - the login time limit in seconds

See Also:

[getLoginTimeout\(\)](#)

getLoginTimeout

```
public static int getLoginTimeout()
```

Gets the maximum time in seconds that a driver can wait when attempting to log in to a database.

Returns:

the driver login time limit in seconds

See Also:

[setLoginTimeout\(int\)](#)

setLogStream

```
public static void setLogStream(PrintStream out)
```

Deprecated.

Sets the logging/tracing `PrintStream` that is used by the `DriverManager` and all drivers.

In the Java 2 SDK, Standard Edition, version 1.3 release, this method checks to see that there is an `SQLPermission` object before setting the logging stream. If a `SecurityManager` exists and its `checkPermission` method denies setting the log writer, this method throws a `java.lang.SecurityException`.

Parameters:

`out` - the new logging/tracing `PrintStream`; to disable, set to `null`

Throws:

[SecurityException](#) - if a security manager exists and its `checkPermission` method denies setting the log stream

See Also:

[SecurityManager.checkPermission\(java.security.Permission\),
getLogStream\(\)](#)

getLogStream

```
public static PrintStream getLogStream()
```

Deprecated.

Retrieves the logging/tracing `PrintStream` that is used by the `DriverManager` and all drivers.

Returns:

the logging/tracing `PrintStream`; if disabled, is `null`

See Also:

[setLogStream\(java.io.PrintStream\)](#)

println

```
public static void println(String message)
```

Prints a message to the current JDBC log stream.

Parameters:

`message` - a log or tracing message

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)*Java™ 2 Platform
Std. Ed. v1.4.2*[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#) [All Classes](#)SUMMARY: NESTED | FIELD | CONSTR | [METHOD](#) DETAIL: FIELD | CONSTR | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

```
import java.sql.DriverManager;
import java.sql.SQLException;
import com.mysql.jdbc.Connection;

public class JDBCCConnect {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;
        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException e1) {
            System.err.println("Error establishing database connection");
            e1.printStackTrace();
        }
    }
}
```

```
import java.sql.Driver;
import java.sql.DriverManager;
import java.util.Enumeration;

public class JDBCdriver {

    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }

        System.out.println(
            "DriverManager:\nlogin timeout=" + DriverManager.getLoginTimeout());

        Enumeration e = DriverManager.getDrivers();
        while (e.hasMoreElements()) {
            Driver drv = (Driver) e.nextElement();

            System.out.println(
                "Driver="
                + drv.getClass().getName()
                + "\nmajor version="
                + drv.getMajorVersion()
                + "\nminor version="
                + drv.getMinorVersion()
                + "\nJDBC compliant="
                + drv.jdbcCompliant());
        }
    }
}
```

DriverManager:

login timeout=0

Driver=com.mysql.jdbc.Driver

major version=3

minor version=0

JDBC compliant=false



The Source for Java Developers



Developers Home > Products & Technologies > Java Technology > J2SE > Core Java
>

Profile and Registration | Why Register?

Core Java

Java Naming and Directory Interface (JNDI)

Downloads

- [Early Access](#)

Reference

- [API Specifications](#)
- [Documentation](#)
- [Code Samples and Apps](#)
- [Technical Articles and Tips](#)
- [Industry Support](#)
- [FAQs](#)

Community

- [User Groups](#)
- [Bookshelf](#)
- [Bug Database](#)
- [Forums](#)

Learning

- [Tutorial and Code Camps](#)

The Java Naming and Directory Interface (JNDI) is a standard extension to the Java platform, providing applications based on Java technology with a unified interface to multiple naming and directory services. You can build powerful and portable directory-enabled applications using this industry standard.

» [Read More](#)

JNDI and J2EE Technologies

JNDI works in concert with other technologies in the [Java 2 Platform, Enterprise Edition \(J2EE\)](#) to organize and locate components in a distributed computing environment.

What's New

May 15 2003

JNDI/LDAP Booster Pack 1.0 Download the booster pack, which contains support for a number of popular LDAP controls and extensions, groups, and Java RMI/CORBA objects. It replaces the booster pack that was bundled with the LDAP 1.2.4 service provider.

November 7, 2002

JNDI Tutorial See the latest version of the JNDI Tutorial.

September 16, 2002

Java 2 SDK, Standard Edition, v 1.4.1 This release of the Java platform includes several [JNDI-related enhancements](#), including support for connection pooling for the LDAP service provider, and automatic discovery of LDAP and DNS services.

September 5, 2002

JNDI/DSML v2 service providers early access release available.

Community

Events

2004 JavaOne Conference. San Francisco , CA Be there!

Join the thousands of developers worldwide who come to the JavaOne conference each year in San Francisco to immerse themselves in Java technology, the latest innovations, the community, and the learning opportunities. » [Read More](#)

Subscribe to Newsletters. Members of [Sun Developer Network](#) can [sign up](#) to receive these (and other) newsletters. Not yet a member? [Join us!](#)

Java Technology Fundamentals

New to Java? Learn the basics of the Java programming language and keep up-to-date on additions to the New-to-Java Programming Center.

Core Java Technologies Newsletter

Find out about new enterprise Java technologies, products, tools, and resources for developers.

Core Java Technologies Tech Tips

Get expert tips, sample code solutions, and techniques for

Related Links

Popular Downloads

- [J2SE 1.4.2](#)
- [Java Web Services Developer Pack 1.3](#)

Technical Topics

- [Performance](#)
- [Web Services](#)
- [Security](#)
- [Desktop](#)

Products and Technologies

- [J2SE 1.5 \(coming soon\)](#)
- [J2SE 1.4.2](#)
- [Desktop Java](#)
- [Java Web Services](#)
- [J2EE](#)
- [Java Dynamic Management Kit \(JDMK\)](#)
- [Sun Java Studio Standard IDE](#)

Sun Resources

- [java.net](#)
- [Java Upgrade Program](#)
- [New to Java Center](#)
- [Professional Certification](#)
- [Professional Training](#)
- [JavaOne Online](#)

developing in the Java 2 Platform, Standard Edition (J2SE).

[» Read More](#)

Feedback

If you have comments, questions, or feedback on JNDI, write to us at jndi@java.sun.com.

We also maintain a "jndi-interest" mailing list open to any discussion about JNDI. To subscribe, send an email message with exactly the following syntax in the message body to listserv@java.sun.com:

```
subscribe jndi-interest your real name
```

To remove yourself from the list, use `signoff` instead of `subscribe`.

Your questions may have already been answered. You can browse an [archive](#) of messages previously sent to the "jndi-interest" list.



Raghavan "Rags" Srinivas,
Staff Engineer, Developer and
Security Expert



[Company Info](#) | [About SDN](#) | [Press](#) | [Contact Us](#) |
[Employment](#)
[How to Buy](#) | [Licensing](#) | [Terms of Use](#) | [Privacy](#) |
[Trademarks](#)

Copyright 1994-2004 Sun Microsystems, Inc.

A Sun Developer Network Site

Unless otherwise licensed, code in all technical manuals herein (including articles, FAQs, samples) is provided under this [License](#).

[XML](#) [Content Feeds](#)

javax.sql

Interface DataSource

public interface **DataSource**

A factory for connections to the physical data source that this **DataSource** object represents. An alternative to the **DriverManager** facility, a **DataSource** object is the preferred means of getting a connection. An object that implements the **DataSource** interface will typically be registered with a naming service based on the Java™ Naming and Directory (JNDI) API.

The **DataSource** interface is implemented by a driver vendor. There are three types of implementations:

1. Basic implementation -- produces a standard **Connection** object
2. Connection pooling implementation -- produces a **Connection** object that will automatically participate in connection pooling. This implementation works with a middle-tier connection pooling manager.
3. Distributed transaction implementation -- produces a **Connection** object that may be used for distributed transactions and almost always participates in connection pooling. This implementation works with a middle-tier transaction manager and almost always with a connection pooling manager.

A **DataSource** object has properties that can be modified when necessary. For example, if the data source is moved to a different server, the property for the server can be changed. The benefit is that because the data source's properties can be changed, any code accessing that data source does not need to be changed.

A driver that is accessed via a **DataSource** object does not register itself with the **DriverManager**. Rather, a **DataSource** object is retrieved through a lookup operation and then used to create a **Connection** object. With a basic implementation, the connection obtained through a **DataSource** object is identical to a connection obtained through the **DriverManager** facility.

Since:

1.4

Method Summary

Connection	getConnection() Attempts to establish a connection with the data source that this <code>DataSource</code> object represents.
Connection	getConnection(String username, String password) Attempts to establish a connection with the data source that this <code>DataSource</code> object represents.
<code>int</code>	getLoginTimeout() Gets the maximum time in seconds that this data source can wait while attempting to connect to a database.
PrintWriter	getLogWriter() Retrieves the log writer for this <code>DataSource</code> object.
<code>void</code>	setLoginTimeout(int seconds) Sets the maximum time in seconds that this data source will wait while attempting to connect to a database.
<code>void</code>	setLogWriter(PrintWriter out) Sets the log writer for this <code>DataSource</code> object to the given <code>java.io.PrintWriter</code> object.

Method Detail

getConnection

```
public Connection getConnection()
           throws SQLException
```

Attempts to establish a connection with the data source that this `DataSource` object represents.

Returns:

a connection to the data source

Throws:

[SQLException](#) - if a database access error occurs

getConnection

```
public Connection getConnection(String username,
                                String password)
                                throws SQLException
```

Attempts to establish a connection with the data source that this `DataSource` object represents.

Parameters:

username - the database user on whose behalf the connection is being made

password - the user's password

Returns:

a connection to the data source

Throws:

[SQLException](#) - if a database access error occurs

getLogWriter

```
public PrintWriter getLogWriter()
                                throws SQLException
```

Retrieves the log writer for this `DataSource` object.

The log writer is a character output stream to which all logging and tracing messages for this data source will be printed. This includes messages printed by the methods of this object, messages printed by methods of other objects manufactured by this object, and so on. Messages printed to a data source specific log writer are not printed to the log writer associated with the `java.sql.DriverManager` class. When a `DataSource` object is created, the log writer is initially null; in other words, the default is for logging to be disabled.

Returns:

the log writer for this data source or null if logging is disabled

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[setLogWriter\(java.io.PrintWriter\)](#)

setLogWriter

```
public void setLogWriter(PrintWriter out)
```

throws [SQLException](#)

Sets the log writer for this `DataSource` object to the given `java.io.PrintWriter` object.

The log writer is a character output stream to which all logging and tracing messages for this data source will be printed. This includes messages printed by the methods of this object, messages printed by methods of other objects manufactured by this object, and so on. Messages printed to a data source- specific log writer are not printed to the log writer associated with the `java.sql.Drivermanager` class. When a `DataSource` object is created the log writer is initially null; in other words, the default is for logging to be disabled.

Parameters:

`out` - the new log writer; to disable logging, set to null

Throws:

[SQLException](#) - if a database access error occurs

See Also:

[getLogWriter\(\)](#)

setLoginTimeout

```
public void setLoginTimeout(int seconds)
                        throws SQLException
```

Sets the maximum time in seconds that this data source will wait while attempting to connect to a database. A value of zero specifies that the timeout is the default system timeout if there is one; otherwise, it specifies that there is no timeout. When a `DataSource` object is created, the login timeout is initially zero.

Parameters:

`seconds` - the data source login time limit

Throws:

[SQLException](#) - if a database access error occurs.

See Also:

[getLoginTimeout\(\)](#)

getLoginTimeout

```
public int getLoginTimeout()
                        throws SQLException
```

Gets the maximum time in seconds that this data source can wait while attempting to connect to a database. A value of zero means that the timeout is the default system timeout if there is one; otherwise, it means that there is no timeout. When a `DataSource` object is created, the login timeout is initially zero.

Returns:

the data source login time limit

Throws:

[SQLException](#) - if a database access error occurs.

See Also:

[setLoginTimeout\(int\)](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | FIELD | CONSTR | [METHOD](#) DETAIL: FIELD | CONSTR | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import com.mysql.jdbc.jdbc2.optional.MysqlDataSource;

public class JDBConnect2Server {

    public static void main(String[] args) {
        Hashtable env = new Hashtable();
        env.put(
            Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
        env.put(Context.PROVIDER_URL, "file:/tmp/registry");

        MysqlDataSource ds = new MysqlDataSource();
        ds.setDatabaseName("jdbctest");
        Context ctx = null;
        try {
            ctx = new InitialContext(env);
        } catch (NamingException ne) {
            ne.printStackTrace();
        }

        try {
            ctx.rebind("jdbc/mySrc", ds);
        } catch (NamingException ne) {
            ne.printStackTrace();
        }
    }
}
```

```
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

public class JDBCConnect2 {
    public static void main(String[] args) {
        Hashtable env = new Hashtable();
        env.put(
            Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
        env.put(Context.PROVIDER_URL, "file:/tmp/registry");
        Context ctx = null;
        try {
            ctx = new InitialContext(env);
        } catch (NamingException ne) {
            ne.printStackTrace();
        }
        DataSource ds = null;
        try {
            ds = (DataSource) ctx.lookup("jdbc/mySrc");
        } catch (NamingException ne) {
            ne.printStackTrace();
        }
        Connection con = null;
        try {
            con = ds.getConnection("mario", "thePassword");
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
    }
}
```

```
import java.sql.DriverManager;
import java.sql.SQLException;
import javax.sql.PooledConnection;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.jdbc2.optional.MysqlPooledConnection;

public class JDBCConnection3 {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException cnfe) {
            System.err.println("Driver class not found");
            cnfe.printStackTrace();
        }
        Connection con = null;
        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException e1) {
            System.err.println("Error establishing database connection");
            e1.printStackTrace();
        }

        PooledConnection pc = new MysqlPooledConnection(con);

        java.sql.Connection con1 = null;
        try {
            con1 = pc.getConnection();
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
    }
}
```

javax.sql

Interface PooledConnection

All Known Subinterfaces:

[XAConnection](#)

public interface **PooledConnection**

An object that provides hooks for connection pool management. A `PooledConnection` object represents a physical connection to a data source. The connection can be recycled rather than being closed when an application is finished with it, thus reducing the number of connections that need to be made.

An application programmer does not use the `PooledConnection` interface directly; rather, it is used by a middle tier infrastructure that manages the pooling of connections.

When an application calls the method `DataSource.getConnection`, it gets back a `Connection` object. If connection pooling is being done, that `Connection` object is actually a handle to a `PooledConnection` object, which is a physical connection.

The connection pool manager, typically the application server, maintains a pool of `PooledConnection` objects. If there is a `PooledConnection` object available in the pool, the connection pool manager returns a `Connection` object that is a handle to that physical connection. If no `PooledConnection` object is available, the connection pool manager calls the `PooledConnection` method `getConnection` to create a new physical connection and returns a handle to it.

When an application closes a connection, it calls the `Connection` method `close`. When connection pooling is being done, the connection pool manager is notified because it has registered itself as a `ConnectionEventListener` object using the `ConnectionPool` method `addConnectionEventListener`. The connection pool manager deactivates the handle to the `PooledConnection` object and returns the `PooledConnection` object to the pool of connections so that it can be used again. Thus, when an application closes its connection, the underlying physical connection is recycled rather than being closed.

The physical connection is not closed until the connection pool manager calls the

PooledConnection method `close`. This method is generally called to have an orderly shutdown of the server or if a fatal error has made the connection unusable.

Since:

1.4

Method Summary

<code>void</code>	<code>addConnectionEventListener</code> (<code>ConnectionEventListener</code> listener) Registers the given event listener so that it will be notified when an event occurs on this <code>PooledConnection</code> object.
<code>void</code>	<code>close</code> () Closes the physical connection that this <code>PooledConnection</code> object represents.
<code>Connection</code>	<code>getConnection</code> () Creates and returns a <code>Connection</code> object that is a handle for the physical connection that this <code>PooledConnection</code> object represents.
<code>void</code>	<code>removeConnectionEventListener</code> (<code>ConnectionEventListener</code> listener) Removes the given event listener from the list of components that will be notified when an event occurs on this <code>PooledConnection</code> object.

Method Detail

getConnection

```
public Connection getConnection()
           throws SQLException
```

Creates and returns a `Connection` object that is a handle for the physical connection that this `PooledConnection` object represents. The connection pool manager calls this method when an application has called the method `DataSource.getConnection` and there are no `PooledConnection` objects available. See the [interface description](#) for more information.

Returns:

a `Connection` object that is a handle to this `PooledConnection` object

Throws:

[SQLException](#) - if a database access error occurs

close

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

Closes the physical connection that this `PooledConnection` object represents. An application never calls this method directly; it is called by the connection pool module, or manager.

See the [interface description](#) for more information.

Throws:

[SQLException](#) - if a database access error occurs

addConnectionEventListener

```
public void addConnectionEventListener  
(ConnectionEventListener listener)
```

Registers the given event listener so that it will be notified when an event occurs on this `PooledConnection` object.

Parameters:

`listener` - a component, usually the connection pool manager, that has implemented the `ConnectionEventListener` interface and wants to be notified when the connection is closed or has an error

See Also:

[removeConnectionEventListener\(javax.sql.
ConnectionEventListener\)](#)

removeConnectionEventListener

```
public void removeConnectionEventListener  
(ConnectionEventListener listener)
```

Removes the given event listener from the list of components that will be notified when an event occurs on this `PooledConnection` object.

Parameters:

`listener` - a component, usually the connection pool manager, that has implemented the `ConnectionEventListener` interface and been registered with this `PooledConnection` object as a listener

See Also:

[addConnectionEventListener\(javax.sql.ConnectionEventListener\)](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

JavaTM 2 Platform

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

Std. Ed. v1.4.2

SUMMARY: NESTED | FIELD | CONSTR | [METHOD](#) DETAIL: FIELD | CONSTR | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

```
import java.sql.DriverManager;
import java.sql.SQLException;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class JDBCCreateTable {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException e1) {
            System.err.println("Error establishing database connection");
            e1.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt = (Statement) con.createStatement();
        } catch (SQLException e2) {
            System.err.println("Error creating SQL-Statement");
            e2.printStackTrace();
        }
        String createTab = new String("CREATE TABLE EMPLOYEE(" +
            "FNAME VARCHAR(10) NOT NULL," +
            "MINIT VARCHAR(1)," +
            "LNAME VARCHAR(10) NOT NULL," +
            "SSN INTEGER(9) NOT NULL," +
            "BDATE DATE," +
            "ADDRESS VARCHAR(30)," +
            "SEX ENUM('M','F')," +
            "SALARY REAL(7,2) UNSIGNED," +
            "SUPERSSN INTEGER(9)," +
            "DNO INTEGER(1));");
        try {
```

```
        System.out.println("result="+stmt.executeUpdate(createTab));
    } catch (SQLException e3) {
        System.err.println("Error creating table EMPLOYEE");
        e3.printStackTrace();
    }
}
```

result=0

```
import java.sql.DriverManager;
import java.sql.SQLException;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class JDBCAAlterTable {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException e1) {
            System.err.println("Error establishing database connection");
            e1.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt = (Statement) con.createStatement();
        } catch (SQLException e2) {
            System.err.println("Error creating SQL-Statement");
            e2.printStackTrace();
        }
        String createTab =
            new String("ALTER TABLE EMPLOYEE ADD PRIMARY KEY (SSN);");
        try {
            System.out.println("result=" + stmt.executeUpdate(createTab));
        } catch (SQLException e3) {
            System.err.println("Error altering table EMPLOYEE");
            e3.printStackTrace();
        }
    }
}
```

```
import java.sql.DriverManager;
import java.sql.SQLException;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class JDBCInsert1 {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException e1) {
            System.err.println("Error establishing database connection");
            e1.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt = (Statement) con.createStatement();
        } catch (SQLException e2) {
            System.err.println("Error creating SQL-Statement");
            e2.printStackTrace();
        }

        try {
            System.out.println("result=" + stmt.executeUpdate("INSERT INTO EMPLOYEE
VALUES('John', 'B', 'Smith', 123456789, '1965-01-09', '731 Fondren, Houston, TX', 'M', 30000,
333445555, 5);"));
            System.out.println("result=" + stmt.executeUpdate("INSERT INTO EMPLOYEE
VALUES('Franklin', 'T', 'Wong', 333445555, '1955-12-08', '638 Voss, Houston, TX', 'M', 40000,
888665555, 5);"));
            System.out.println("result=" + stmt.executeUpdate("INSERT INTO EMPLOYEE
VALUES('Alicia', 'J', 'Zelaya', 999887777, '1968-07-19', '3321 Castle, Spring, TX', 'F', 25000,
987654321, 4);"));
            System.out.println("result=" + stmt.executeUpdate("INSERT INTO EMPLOYEE
```



```
VALUES('Jennifer', 'S', 'Wallace', 987654321, '1941-06-20', '291 Berry, Bellaire, TX', 'F', 43000,
888665555, 4;));
    System.out.println("result=" + stmt.executeUpdate("INSERT INTO EMPLOYEE
VALUES('Ramesh', 'K', 'Narayan', 666884444, '1962-09-15', '975 Fire Oak, Humble, TX', 'M',
38000, 333445555, 5;));
    System.out.println("result=" + stmt.executeUpdate("INSERT INTO EMPLOYEE
VALUES('Joyce', 'A', 'English', 453453453, '1972-07-31', '5631 Rice, Houston, TX', 'F', 25000,
333445555, 5;));
    System.out.println("result=" + stmt.executeUpdate("INSERT INTO EMPLOYEE
VALUES('Ahmad', 'V', 'Jabbar', 987987987, '1969-03-29', '980 Dallas, Houston, TX', 'M', 25000,
987654321, 4;));
    System.out.println("result=" + stmt.executeUpdate("INSERT INTO EMPLOYEE
VALUES('James', 'E', 'Borg', 888665555, '1937-11-10', '450 Stone, Houston, TX', 'M', 55000, null,
1;));
    } catch (SQLException e3) {
        System.err.println("Error inserting values into table EMPLOYEE");
        e3.printStackTrace();
    }
}
```

result=1
result=1
result=1
result=1
result=1
result=1
result=1
result=1

```
import java.sql.DriverManager;
import java.sql.SQLException;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class JDBCInsert2 {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException e1) {
            System.err.println("Error establishing database connection");
            e1.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt = (Statement) con.createStatement();
        } catch (SQLException e2) {
            System.err.println("Error creating SQL-Statement");
            e2.printStackTrace();
        }

        try {
            stmt.addBatch("INSERT INTO EMPLOYEE VALUES('John', 'B', 'Smith',
123456789, '1965-01-09', '731 Fondren, Houston, TX', 'M', 30000, 333445555, 5);");
            stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Franklin', 'T', 'Wong',
333445555, '1955-12-08', '638 Voss, Houston, TX', 'M', 40000, 888665555, 5);");
            stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Alicia', 'J', 'Zelaya',
999887777, '1968-07-19', '3321 Castle, Spring, TX', 'F', 25000, 987654321, 4);");
            stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Jennifer', 'S', 'Wallace',
987654321, '1941-06-20', '291 Berry, Bellaire, TX', 'F', 43000, 888665555, 4);");
            stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Ramesh', 'K', 'Narayan',
666884444, '1962-09-15', '975 Fire Oak, Humble, TX', 'M', 38000, 333445555, 5);");
```

```
        stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Joyce', 'A', 'English',
453453453, '1972-07-31', '5631 Rice, Houston, TX', 'F', 25000, 333445555, 5);");
        stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Ahmad', 'V', 'Jabbar',
987987987, '1969-03-29', '980 Dallas, Houston, TX', 'M', 25000, 987654321, 4);");
        stmt.addBatch("INSERT INTO EMPLOYEE VALUES('James', 'E', 'Borg',
888665555, '1937-11-10', '450 Stone, Houston, TX', 'M', 55000, null, 1);");
        int[] insertCounts = stmt.executeBatch();
    } catch (SQLException e3) {
        System.err.println("Error inserting values into table EMPLOYEE");
        e3.printStackTrace();
    }
}
```

```
import java.sql.DriverManager;
import java.sql.SQLException;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class JDBCUpdate1 {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException e1) {
            System.err.println("Error establishing database connection");
            e1.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt = (Statement) con.createStatement();
        } catch (SQLException e2) {
            System.err.println("Error creating SQL-Statement");
            e2.printStackTrace();
        }

        try {
            stmt.addBatch("ALTER TABLE EMPLOYEE ADD BDAY VARCHAR(10);");
            stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Sunday' WHERE
DAYOFWEEK(BDATE)=1;");
            stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Monday' WHERE
DAYOFWEEK(BDATE)=2;");
            stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Tuesday' WHERE
DAYOFWEEK(BDATE)=3;");
            stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Wednesday' WHERE
DAYOFWEEK(BDATE)=4;");
            stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Thursday' WHERE
```

```
DAYOFWEEK(BDATE)=5;");
        stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Friday' WHERE DAYOFWEEK
(BDATE)=6;");
        stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Saturday' WHERE
DAYOFWEEK(BDATE)=7;");
        int[] result = stmt.executeBatch();
        for (int i=0; i<result.length;i++){
            System.out.println("Statement No "+i+" changed "+result[i]+" rows");
        }
    } catch (SQLException e3) {
        System.err.println("Error inserting values into table EMPLOYEE");
        e3.printStackTrace();
    }
}
```

Statement No 0 changed 8 rows
Statement No 1 changed 0 rows
Statement No 2 changed 1 rows
Statement No 3 changed 0 rows
Statement No 4 changed 1 rows
Statement No 5 changed 1 rows
Statement No 6 changed 2 rows
Statement No 7 changed 3 rows

```
import java.sql.DriverManager;
import java.sql.SQLException;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.PreparedStatement;
import com.mysql.jdbc.Statement;

public class JDBCUpdate2 {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException e1) {
            System.err.println("Error establishing database connection");
            e1.printStackTrace();
        }

        Statement stmt = null;
        PreparedStatement pstmt = null;
        try {
            stmt = (Statement) con.createStatement();
            pstmt = (PreparedStatement) con.prepareStatement("UPDATE EMPLOYEE SET
BDAY=? WHERE DAYOFWEEK(BDATE)=?;");

        } catch (SQLException e2) {
            System.err.println("Error creating SQL-Statement");
            e2.printStackTrace();
        }

        try {
            String[] days=
{"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"} ;
            stmt.addBatch("ALTER TABLE EMPLOYEE ADD BDAY VARCHAR(10);");
            for (int i=1; i<8;i++){
                pstmt.setString(1,days[i-1]);
```



```
        pstmt.setInt(2,i);
        pstmt.addBatch();
    }
    int[] result = stmt.executeBatch();
    for (int i=0; i<result.length;i++){
        System.out.println("Statement No "+i+" changed "+result[i]+" rows");
    }
} catch (SQLException e3) {
    System.err.println("Error inserting values into table EMPLOYEE");
    e3.printStackTrace();
}
}
```

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class JDBCSelect1 {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException e1) {
            System.err.println("Error establishing database connection");
            e1.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt = (Statement) con.createStatement();
        } catch (SQLException e2) {
            System.err.println("Error creating SQL-Statement");
            e2.printStackTrace();
        }

        try {
            ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
            ResultSetMetaData rsmd = rs.getMetaData();
            int noColumns = rsmd.getColumnCount();
            for (int i = 1; i < noColumns; i++) {
                System.out.print(rsmd.getColumnLabel(i) + "\t");
            }
            System.out.println();
        }
    }
}
```

```
        while (rs.isLast() == false) {
            rs.next();
            for (int i = 1; i < noColumns; i++) {
                System.out.print( rs.getObject(i)+"\t" );
            }
            System.out.println();
        }

    } catch (SQLException e3) {
        System.err.println("Error selecting values from table EMPLOYEE");
        e3.printStackTrace();
    }
}
```

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000.0		
333445555	5								
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000.0		
888665555	5								
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000.0		
987654321	4								
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000.0		
888665555	4								
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M			
38000.0	333445555	5							
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000.0		
333445555	5								
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000.0		
987654321	4								
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000.0	null	1

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class JDBCSelect5 {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException sqle) {
            System.err.println("Error establishing database connection");
            sqle.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt =
                (Statement) con.createStatement();
        } catch (SQLException e2) {
            System.err.println("Error creating SQL-Statement");
            e2.printStackTrace();
        }

        try {
            ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
            rs.afterLast();
            while (rs.previous()){
                System.out.println(rs.getString("FNAME"));
            }
        } catch (SQLException sqle) {
            System.err.println("Error selecting values from table EMPLOYEE");
            sqle.printStackTrace();
        }
    }
}
```

```
    }  
    }  
}
```

James
Ahmad
Joyce
Ramesh
Jennifer
Alicia
Franklin
John

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class JDBCSelect6 {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException sqle) {
            System.err.println("Error establishing database connection");
            sqle.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt = (Statement) con.createStatement();
        } catch (SQLException e2) {
            System.err.println("Error creating SQL-Statement");
            e2.printStackTrace();
        }

        try {
            int position = 0;
            ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
            rs.last();
            int size = rs.getRow();
            for (int i = 0; i < size; i++) {
                position = (position + 3) % size;
                rs.absolute(position + 1);
                System.out.println(
                    "position=" + (position + 1) + ": " + rs.getString("FNAME"));
            }
        }
    }
}
```



```
        }  
    } catch (SQLException sqle) {  
        System.err.println("Error selecting values from table EMPLOYEE");  
        sqle.printStackTrace();  
    }  
}  
}
```

position=4: Jennifer
position=7: Ahmad
position=2: Franklin
position=5: Ramesh
position=8: James
position=3: Alicia
position=6: Joyce
position=1: John

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

import com.mysql.jdbc.Connection;

public class JDBCSelect2 {
    private static void printResultSet(ResultSet rs) throws SQLException {
        ResultSetMetaData rsmd = rs.getMetaData();
        int noColumns = rsmd.getColumnCount();
        for (int i = 1; i < noColumns; i++) {
            System.out.print(rsmd.getColumnLabel(i) + "\t");
        }
        System.out.println();

        while (rs.isLast() == false) {
            rs.next();
            for (int i = 1; i < noColumns; i++) {
                System.out.print( rs.getObject(i)+"\t" );
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException e1) {
            System.err.println("Error establishing database connection");
            e1.printStackTrace();
        }
    }
}
```

```
Statement stmt = null;
```

```
try {
```

```
    stmt = (Statement) con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,  
ResultSet.CONCUR_UPDATABLE);
```

```
    } catch (SQLException e2) {
```

```
        System.err.println("Error creating SQL-Statement");
```

```
        e2.printStackTrace();
```

```
    }
```

```
try {
```

```
    ResultSet uprs = (ResultSet) stmt.executeQuery("SELECT * FROM EMPLOYEE;");
```

```
    printResultSet(uprs);
```

```
    uprs.moveToInsertRow();
```

```
    uprs.updateString("FNAME","Mario");
```

```
    uprs.updateString("LNAME","Jeckle");
```

```
    uprs.updateInt("SSN",111111111);
```

```
    uprs.insertRow();
```

```
    uprs = (ResultSet) stmt.executeQuery("SELECT * FROM EMPLOYEE;");
```

```
    printResultSet(uprs);
```

```
    } catch (SQLException e3) {
```

```
        System.err.println("Error selecting values from table EMPLOYEE");
```

```
        e3.printStackTrace();
```

```
    }
```

```
}
```

```
}
```

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000.0		
333445555	5								
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000.0		
888665555	5								
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000.0		
333445555	5								
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M			
38000.0	333445555	5							
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000.0	null	1
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000.0		
888665555	4								
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000.0		
987654321	4								
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000.0		
987654321	4								
FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Mario	null	Jeckle	111111111	null	null	null	null		
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000.0		
333445555	5								
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000.0		
888665555	5								
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000.0		
333445555	5								
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M			
38000.0	333445555	5							
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000.0	null	1
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000.0		
888665555	4								
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000.0		
987654321	4								
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000.0		
987654321	4								

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.mysql.jdbc.Connection;

public class JDBCSelect3 {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException e1) {
            System.err.println("Error establishing database connection");
            e1.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt =
                (Statement) con.createStatement(
                    ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
        } catch (SQLException e2) {
            System.err.println("Error creating SQL-Statement");
            e2.printStackTrace();
        }

        try {
            ResultSet uprs =
                (ResultSet) stmt.executeQuery("SELECT * FROM EMPLOYEE;");
            int namePos = uprs.findColumn("LNAME");

            while (uprs.isLast() == false) {
```

```
        uprs.next();
        if (uprs.getString(namePos).compareTo("Wallace") == 0) {
            uprs.updateString(namePos, "Doe");
            uprs.updateRow();
        }
    }

} catch (SQLException e3) {
    System.err.println("Error selecting values from table EMPLOYEE");
    e3.printStackTrace();
}

}
```

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.mysql.jdbc.Connection;

public class JDBCSelect4 {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException e1) {
            System.err.println("Error establishing database connection");
            e1.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt =
                (Statement) con.createStatement(
                    ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
        } catch (SQLException e2) {
            System.err.println("Error creating SQL-Statement");
            e2.printStackTrace();
        }

        try {
            ResultSet uprs =
                (ResultSet) stmt.executeQuery("SELECT * FROM EMPLOYEE;");
            int namePos = uprs.findColumn("LNAME");

            while (uprs.isLast() == false) {
```



```
        uprs.next();
        if (uprs.getString(namePos).compareTo("Smith") == 0) {
            uprs.deleteRow();
        }
    }

} catch (SQLException e3) {
    System.err.println("Error selecting values from table EMPLOYEE");
    e3.printStackTrace();
}
}
```

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class JDBCSelect7 {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException cnfe) {
            System.err.println("Driver class not found");
            cnfe.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException sqle) {
            System.err.println("Error establishing database connection");
            sqle.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt =
                (Statement) con.createStatement(
                    ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
        } catch (SQLException sqle) {
            System.err.println("Error creating SQL-Statement");
            sqle.printStackTrace();
        }

        try {
            ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
            rs.absolute(5);
            System.out.println(rs.getString("FNAME"));

            System.out.println("sleeping ...");
            Thread.sleep(6000);
        }
    }
}
```

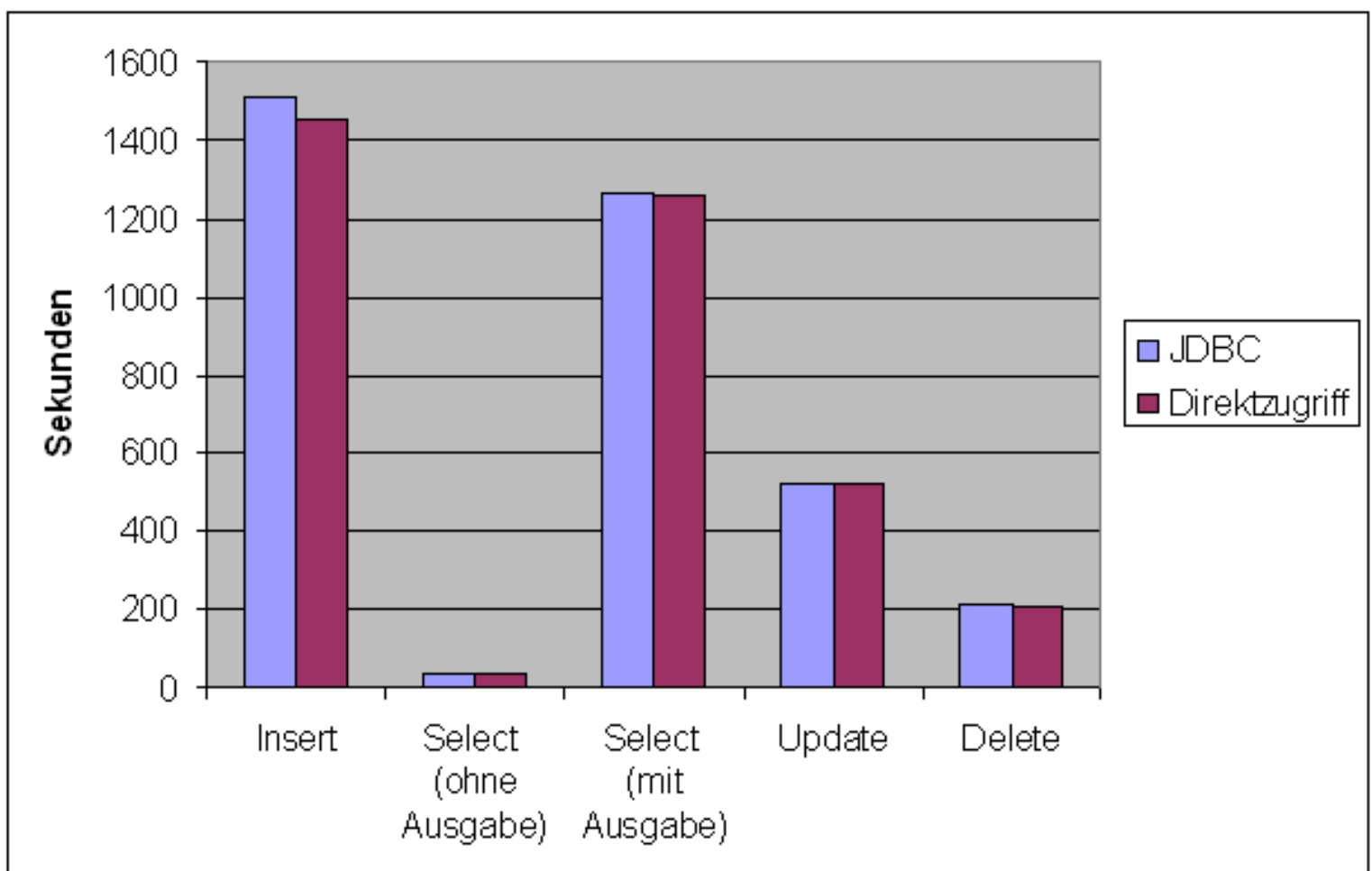
```
System.out.println("awake ...");
```

```
if (rs.rowUpdated() == true) {  
    rs.refreshRow();  
    System.out.println(rs.getString("FNAME"));  
}
```

```
} catch (SQLException sqle) {  
    System.err.println("Error selecting values from table EMPLOYEE");  
    sqle.printStackTrace();  
} catch (InterruptedException ie) {  
    ie.printStackTrace();  
}
```

```
}
```

```
}
```



Network Working Group	Paul J. Leach, Microsoft
INTERNET-DRAFT	Rich Salz, Certco
<draft-leach-uuids-guids-01.txt>	
Category: Standards Track	
Expires August 4, 1998	February 4, 1998

UUIDs and GUIDs

STATUS OF THIS MEMO

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress".

To learn the current status of any Internet-Draft, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on [ftp.is.co.za](ftp://ftp.is.co.za) (Africa), [nic.nordu.net](ftp://nic.nordu.net) (Europe), [munnari.oz.au](ftp://munnari.oz.au) (Pacific Rim), [ds.internic.net](ftp://ds.internic.net) (US East Coast), or [ftp.isi.edu](ftp://ftp.isi.edu) (US West Coast).

Distribution of this document is unlimited. Please send comments to the authors or the CIFS mailing list at [<cifs@discuss.microsoft.com>](mailto:cifs@discuss.microsoft.com). Discussions of the mailing list are archived at [<URL:http://discuss.microsoft.com/archives/index>](http://discuss.microsoft.com/archives/index).

ABSTRACT

This specification defines the format of UUIDs (Universally Unique Identifier), also known as GUIDs (Globally Unique Identifier). A UUID is 128 bits long, and if generated according to the one of the mechanisms in this document, is either guaranteed to be different

from all other UUIDs/GUIDs generated until 3400 A.D. or extremely likely to be different (depending on the mechanism chosen). UUIDs were originally used in the Network Computing System (NCS) [1] and later in the Open Software Foundation's (OSF) Distributed Computing Environment [2].

This specification is derived from the latter specification with the kind permission of the OSF.

Table of Contents

1. Introduction	3

[Page 1]

Internet-Draft	UUIDs and GUIDs (DRAFT)	02/04/98

2. Motivation	3

3. Specification	3

3.1 Format.....	4

3.1.1 Variant.....	4

3.1.2 UUID layout.....	5

3.1.3 Version.....	5

3.1.4 Timestamp.....	6

3.1.5 Clock sequence.....	6

3.1.6 Node.....	7

3.1.7 Nil UUID.....	7

3.2 Algorithms for creating a time-based UUID.....	7

3.2.1 Basic algorithm.....	7

3.2.2	Reading stable storage.....	8
3.2.3	System clock resolution.....	8
3.2.4	Writing stable storage.....	9
3.2.5	Sharing state across processes.....	9
3.2.6	UUID Generation details.....	9
3.3	Algorithm for creating a name-based UUID.....	10
3.4	Algorithms for creating a UUID from truly random or pseudo-random numbers	11
3.5	String Representation of UUIDs.....	12
3.6	Comparing UUIDs for equality.....	12
3.7	Comparing UUIDs for relative order.....	13
3.8	Byte order of UUIDs.....	13
4.	Node IDs when no IEEE 802 network card is available	14
5.	Obtaining IEEE 802 addresses	15
6.	Security Considerations	15
7.	Acknowledgements	15
Leach, Salz	expires Aug 1998	[Page 2]
Internet-Draft	UUIDs and GUIDs (DRAFT)	02/04/98
8.	References	15
9.	Authors' addresses	16
10.	Notice	16
11.	Full Copyright Statement	16

Appendix A _ UUID Sample Implementation.....	17
Appendix B _ Sample output of utest.....	27
Appendix C _ Some name space IDs.....	27

1. Introduction

This specification defines the format of UUIDs (Universally Unique IDentifiers), also known as GUIDs (Globally Unique IDentifiers). A UUID is 128 bits long, and if generated according to the one of the mechanisms in this document, is either guaranteed to be different from all other UUIDs/GUIDs generated until 3400 A.D. or extremely likely to be different (depending on the mechanism chosen).

2. Motivation

One of the main reasons for using UUIDs is that no centralized authority is required to administer them (beyond the one that allocates IEEE 802.1 node identifiers). As a result, generation on demand can be completely automated, and they can be used for a wide variety of purposes. The UUID generation algorithm described here supports very high allocation rates: 10 million per second per machine if you need it, so that they could even be used as transaction IDs.

UUIDs are fixed-size (128-bits) which is reasonably small relative to other alternatives. This fixed, relatively small size lends itself well to sorting, ordering, and hashing of all sorts, storing in databases, simple allocation, and ease of programming in general.

3. Specification

A UUID is an identifier that is unique across both space and time, with respect to the space of all UUIDs. To be precise, the UUID consists of a finite bit space. Thus the time value used for constructing a UUID is limited and will roll over in the future (approximately at A.D. 3400, based on the specified algorithm). A UUID can be used for multiple purposes, from tagging objects with an extremely short lifetime, to reliably identifying very persistent objects across a network.

The generation of UUIDs does not require that a registration authority be contacted for each identifier. Instead, it requires a unique value over space for each UUID generator. This spatially unique value is specified as an IEEE 802 address, which is usually already available to network-connected systems. This 48-bit address can be assigned based on an address block obtained through the IEEE registration authority. This section of the UUID specification assumes the availability of an IEEE 802 address to a system desiring to generate a UUID, but if one is not available section 4 specifies a way to generate a probabilistically unique one that can not conflict with any properly assigned IEEE 802 address.

3.1 Format

In its most general form, all that can be said of the UUID format is that a UUID is 16 octets, and that some bits of octet 8 of the UUID called the variant field (specified in the next section) determine finer structure.

3.1.1 Variant

The variant field determines the layout of the UUID. That is, the interpretation of all other bits in the UUID depends on the setting of the bits in the variant field. The variant field consists of a variable number of the msbs of octet 8 of the UUID.

The following table lists the contents of the variant field.

Msb0	Msb1	Msb2	Description

0	-	-	Reserved, NCS backward compatibility.

1	0	-	The variant specified in this document.

1	1	0	Reserved, Microsoft Corporation backward compatibility

1 1 1 Reserved for future definition.

Other UUID variants may not interoperate with the UUID variant specified in this document, where interoperability is defined as the applicability of operations such as string conversion and lexical ordering across different systems. However, UUIDs allocated according to the stricture of different variants, though they may define different interpretations of the bits outside the variant field, will not result in duplicate UUID allocation, because of the differing values of the variant field itself.

The remaining fields described below (version, timestamp, etc.) are defined only for the UUID variant noted above.

Leach, Salz expires Aug 1998 [Page 4]

Internet-Draft UUIDs and GUIDs (DRAFT) 02/04/98

3.1.2 UUID layout

The following table gives the format of a UUID for the variant specified herein. The UUID consists of a record of 16 octets. To minimize confusion about bit assignments within octets, the UUID record definition is defined only in terms of fields that are integral numbers of octets. The fields are in order of significance for comparison purposes, with "time_low" the most significant, and "node" the least significant.

Field	Data Type	Octet #	Note
time_low	unsigned 32 bit integer	0-3	The low field of the timestamp.
time_mid	unsigned 16 bit integer	4-5	The middle field of the timestamp.
time_hi_and_version	unsigned 16 bit integer	6-7	The high field of the timestamp multiplexed with the version number.

clock_seq_hi_and_rese unsigned 8 8 The high field of the
rved bit integer clock sequence
multiplexed with the
variant.

clock_seq_low unsigned 8 9 The low field of the
bit integer clock sequence.

node unsigned 48 10-15 The spatially unique
bit integer node identifier.

3.1.3 Version

The version number is in the most significant 4 bits of the time
stamp (time_hi_and_version).

The following table lists currently defined versions of the UUID.

Msb0	Msb1	Msb2	Msb3	Version	Description
0	0	0	1	1	The time-based version specified in this document.
0	0	1	0	2	Reserved for DCE Security version, with embedded POSIX UIDs.
0	0	1	1	3	The name-based version specified in this

Leach, Salz expires Aug 1998 [Page 5]

Internet-Draft UUIDs and GUIDs (DRAFT) 02/04/98

document

0	1	0	0	4	The randomly or pseudo- randomly generated version specified in this document

3.1.4 Timestamp

The timestamp is a 60 bit value. For UUID version 1, this is represented by Coordinated Universal Time (UTC) as a count of 100-nanosecond intervals since 00:00:00.00, 15 October 1582 (the date of Gregorian reform to the Christian calendar).

For systems that do not have UTC available, but do have local time, they MAY use local time instead of UTC, as long as they do so consistently throughout the system. This is NOT RECOMMENDED, however, and it should be noted that all that is needed to generate UTC, given local time, is a time zone offset.

For UUID version 3, it is a 60 bit value constructed from a name.

For UUID version 4, it is a randomly or pseudo-randomly generated 60 bit value.

3.1.5 Clock sequence

For UUID version 1, the clock sequence is used to help avoid duplicates that could arise when the clock is set backwards in time or if the node ID changes.

If the clock is set backwards, or even might have been set backwards (e.g., while the system was powered off), and the UUID generator can not be sure that no UUIDs were generated with timestamps larger than the value to which the clock was set, then the clock sequence has to be changed. If the previous value of the clock sequence is known, it can be just incremented; otherwise it should be set to a random or high-quality pseudo random value.

Similarly, if the node ID changes (e.g. because a network card has been moved between machines), setting the clock sequence to a random number minimizes the probability of a duplicate due to slight differences in the clock settings of the machines. (If the value of clock sequence associated with the changed node ID were known, then the clock sequence could just be incremented, but that is unlikely.)

The clock sequence MUST be originally (i.e., once in the lifetime of a system) initialized to a random number to minimize the correlation across systems. This provides maximum protection against node identifiers that may move or switch from system to system rapidly. The initial value MUST NOT be correlated to the node identifier.

For UUID version 3, it is a 14 bit value constructed from a name.

Leach, Salz

expires Aug 1998

[Page 6]

Internet-Draft

UUIDs and GUIDs (DRAFT)

02/04/98

For UUID version 4, it is a randomly or pseudo-randomly generated 14 bit value.

3.1.6 Node

For UUID version 1, the node field consists of the IEEE address, usually the host address. For systems with multiple IEEE 802 addresses, any available address can be used. The lowest addressed octet (octet number 10) contains the global/local bit and the unicast/multicast bit, and is the first octet of the address transmitted on an 802.3 LAN.

For systems with no IEEE address, a randomly or pseudo-randomly generated value may be used (see section 4). The multicast bit must be set in such addresses, in order that they will never conflict with addresses obtained from network cards.

For UUID version 3, the node field is a 48 bit value constructed from a name.

For UUID version 4, the node field is a randomly or pseudo-randomly generated 48 bit value.

3.1.7 Nil UUID

The nil UUID is special form of UUID that is specified to have all 128 bits set to 0 (zero).

3.2 Algorithms for creating a time-based UUID

Various aspects of the algorithm for creating a version 1 UUID are discussed in the following sections. UUID generation requires a guarantee of uniqueness within the node ID for a given variant and version. Interoperability is provided by complying with the specified data structure.

3.2.1 Basic algorithm

The following algorithm is simple, correct, and inefficient:

- . Obtain a system wide global lock
- . From a system wide shared stable store (e.g., a file), read the UUID generator state: the values of the time stamp, clock sequence, and node ID used to generate the last UUID.
- . Get the current time as a 60 bit count of 100-nanosecond intervals since 00:00:00.00, 15 October 1582
- . Get the current node ID

Leach, Salz

expires Aug 1998

[Page 7]

Internet-Draft

UUIDs and GUIDs (DRAFT)

02/04/98

- . If the state was unavailable (non-existent or corrupted), or the saved node ID is different than the current node ID, generate a random clock sequence value
- . If the state was available, but the saved time stamp is later than the current time stamp, increment the clock sequence value
- . Format a UUID from the current time stamp, clock sequence, and node ID values according to the structure in section 3.1 (see section 3.2.6 for more details)
- . Save the state (current time stamp, clock sequence, and node ID) back to the stable store
- . Release the system wide global lock

If UUIDs do not need to be frequently generated, the above algorithm may be perfectly adequate. For higher performance requirements, however, issues with the basic algorithm include:

- . Reading the state from stable storage each time is inefficient
- . The resolution of the system clock may not be 100-nanoseconds
- . Writing the state to stable storage each time is inefficient
- . Sharing the state across process boundaries may be inefficient

Each of these issues can be addressed in a modular fashion by local improvements in the functions that read and write the state and read the clock. We address each of them in turn in the following sections.

3.2.2 Reading stable storage

The state only needs to be read from stable storage once at boot time, if it is read into a system wide shared volatile store (and updated whenever the stable store is updated).

If an implementation does not have any stable store available, then it can always say that the values were unavailable. This is the least desirable implementation, because it will increase the frequency of creation of new clock sequence numbers, which increases the probability of duplicates.

If the node ID can never change (e.g., the net card is inseparable from the system), or if any change also reinitializes the clock sequence to a random value, then instead of keeping it in stable store, the current node ID may be returned.

3.2.3 System clock resolution

The time stamp is generated from the system time, whose resolution may be less than the resolution of the UUID time stamp.

Leach, Salz expires Aug 1998 [Page 8]

Internet-Draft UUIDs and GUIDs (DRAFT) 02/04/98

If UUIDs do not need to be frequently generated, the time stamp can simply be the system time multiplied by the number of 100-nanosecond intervals per system time interval.

If a system overruns the generator by requesting too many UUIDs within a single system time interval, the UUID service **MUST** either: return an error, or stall the UUID generator until the system clock catches up.

A high resolution time stamp can be simulated by keeping a count of how many UUIDs have been generated with the same value of the system time, and using it to construction the low-order bits of the time stamp. The count will range between zero and the number of 100-nanosecond intervals per system time interval.

Note: if the processors overrun the UUID generation frequently, additional node identifiers can be allocated to the system, which will permit higher speed allocation by making multiple UUIDs potentially available for each time stamp value.

3.2.4 Writing stable storage

The state does not always need to be written to stable store every time a UUID is generated. The timestamp in the stable store can be periodically set to a value larger than any yet used in a UUID; as long as the generated UUIDs have time stamps less than that value, and the clock sequence and node ID remain unchanged, only the shared volatile copy of the state needs to be updated. Furthermore, if the time stamp value in stable store is in the future by less than the typical time it takes the system to reboot, a crash will not cause a reinitialization of the clock sequence.

3.2.5 Sharing state across processes

If it is too expensive to access shared state each time a UUID is generated, then the system wide generator can be implemented to allocate a block of time stamps each time it is called, and a per-process generator can allocate from that block until it is exhausted.

3.2.6 UUID Generation details

UUIDs are generated according to the following algorithm:

- Determine the values for the UTC-based timestamp and clock sequence to be used in the UUID, as described above.
- For the purposes of this algorithm, consider the timestamp to be a 60-bit unsigned integer and the clock sequence to be a 14-bit unsigned integer. Sequentially number the bits in a field, starting from 0 (zero) for the least significant bit.

- Set the time_low field equal to the least significant 32-bits (bits numbered 0 to 31 inclusive) of the time stamp in the same order of significance.

Leach, Salz

expires Aug 1998

[Page 9]

Internet-Draft

UUIDs and GUIDs (DRAFT)

02/04/98

- Set the time_mid field equal to the bits numbered 32 to 47 inclusive of the time stamp in the same order of significance.
- Set the 12 least significant bits (bits numbered 0 to 11 inclusive) of the time_hi_and_version field equal to the bits numbered 48 to 59 inclusive of the time stamp in the same order of significance.
- Set the 4 most significant bits (bits numbered 12 to 15 inclusive) of the time_hi_and_version field to the 4-bit version number corresponding to the UUID version being created, as shown in the table in section 3.1.3.
- Set the clock_seq_low field to the 8 least significant bits (bits numbered 0 to 7 inclusive) of the clock sequence in the same order of significance.
- Set the 6 least significant bits (bits numbered 0 to 5 inclusive) of the clock_seq_hi_and_reserved field to the 6 most significant bits (bits numbered 8 to 13 inclusive) of the clock sequence in the same order of significance.
- Set the 2 most significant bits (bits numbered 6 and 7) of the clock_seq_hi_and_reserved to 0 and 1, respectively.
- Set the node field to the 48-bit IEEE address in the same order of significance as the address.

3.3 Algorithm for creating a name-based UUID

The version 3 UUID is meant for generating UUIDs from "names" that are drawn from, and unique within, some "name space". Some examples of names (and, implicitly, name spaces) might be DNS names, URLs, ISO Object IDs (OIDs), reserved words in a programming language, or X.500

Distinguished Names (DNs); thus, the concept of name and name space should be broadly construed, and not limited to textual names. The mechanisms or conventions for allocating names from, and ensuring their uniqueness within, their name spaces are beyond the scope of this specification.

The requirements for such UUIDs are as follows:

- . The UUIDs generated at different times from the same name in the same namespace **MUST** be equal
- . The UUIDs generated from two different names in the same namespace should be different (with very high probability)
- . The UUIDs generated from the same name in two different namespaces should be different with (very high probability)
- . If two UUIDs that were generated from names are equal, then they were generated from the same name in the same namespace (with very high probability).

Leach, Salz expires Aug 1998 [Page 10]

Internet-Draft UUIDs and GUIDs (DRAFT) 02/04/98

The algorithm for generating the a UUID from a name and a name space are as follows:

- . Allocate a UUID to use as a "name space ID" for all UUIDs generated from names in that name space
- . Convert the name to a canonical sequence of octets (as defined by the standards or conventions of its name space); put the name space ID in network byte order
- . Compute the MD5 [3] hash of the name space ID concatenated with the name
- . Set octets 0-3 of time_low field to octets 0-3 of the MD5 hash
- . Set octets 0-1 of time_mid field to octets 4-5 of the MD5 hash
- . Set octets 0-1 of time_hi_and_version field to octets 6-7 of the

MD5 hash

- . Set the clock_seq_hi_and_reserved field to octet 8 of the MD5 hash
- . Set the clock_seq_low field to octet 9 of the MD5 hash
- . Set octets 0-5 of the node field to octets 10-15 of the MD5 hash
- . Set the 2 most significant bits (bits numbered 6 and 7) of the clock_seq_hi_and_reserved to 0 and 1, respectively.
- . Set the 4 most significant bits (bits numbered 12 to 15 inclusive) of the time_hi_and_version field to the 4-bit version number corresponding to the UUID version being created, as shown in the table above.
- . Convert the resulting UUID to local byte order.

3.4 Algorithms for creating a UUID from truly random or pseudo-random numbers

The version 4 UUID is meant for generating UUIDs from truly-random or pseudo-random numbers.

The algorithm is as follows:

- . Set the 2 most significant bits (bits numbered 6 and 7) of the clock_seq_hi_and_reserved to 0 and 1, respectively.
- . Set the 4 most significant bits (bits numbered 12 to 15 inclusive) of the time_hi_and_version field to the 4-bit version number corresponding to the UUID version being created, as shown in the table above.

Leach, Salz

expires Aug 1998

[Page 11]

Internet-Draft

UUIDs and GUIDs (DRAFT)

02/04/98

- . Set all the other bits to randomly (or pseudo-randomly) chosen values.

Here are several possible ways to generate the random values:

- . Use a physical source of randomness: for example, a white noise generator, radioactive decay, or a lava lamp.
- . Use a cryptographic strength random number generator.

3.5 String Representation of UUIDs

For use in human readable text, a UUID string representation is specified as a sequence of fields, some of which are separated by single dashes.

Each field is treated as an integer and has its value printed as a zero-filled hexadecimal digit string with the most significant digit first. The hexadecimal values a to f inclusive are output as lower case characters, and are case insensitive on input. The sequence is the same as the UUID constructed type.

The formal definition of the UUID string representation is provided by the following extended BNF:

```
UUID          = <time_low> "-" <time_mid> "-"
               <time_high_and_version> "-"
               <clock_seq_and_reserved>
               <clock_seq_low> "-" <node>
time_low      = 4*<hexOctet>
time_mid      = 2*<hexOctet>
time_high_and_version = 2*<hexOctet>
clock_seq_and_reserved = <hexOctet>
clock_seq_low = <hexOctet>
node          = 6*<hexOctet>
hexOctet      = <hexDigit> <hexDigit>
hexDigit =
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
    | "a" | "b" | "c" | "d" | "e" | "f"
    | "A" | "B" | "C" | "D" | "E" | "F"
```

The following is an example of the string representation of a UUID:

f81d4fae-7dec-11d0-a765-00a0c91e6bf6

3.6 Comparing UUIDs for equality

Consider each field of the UUID to be an unsigned integer as shown in the table in section 3.1. Then, to compare a pair of UUIDs, arithmetically compare the corresponding fields from each UUID in order of significance and according to their data type. Two UUIDs are equal if and only if all the corresponding fields are equal.

Leach, Salz expires Aug 1998 [Page 12]

Internet-Draft UUIDs and GUIDs (DRAFT) 02/04/98

Note: as a practical matter, on many systems comparison of two UUIDs for equality can be performed simply by comparing the 128 bits of their in-memory representation considered as a 128 bit unsigned integer. Here, it is presumed that by the time the in-memory representation is obtained the appropriate byte-order canonicalizations have been carried out.

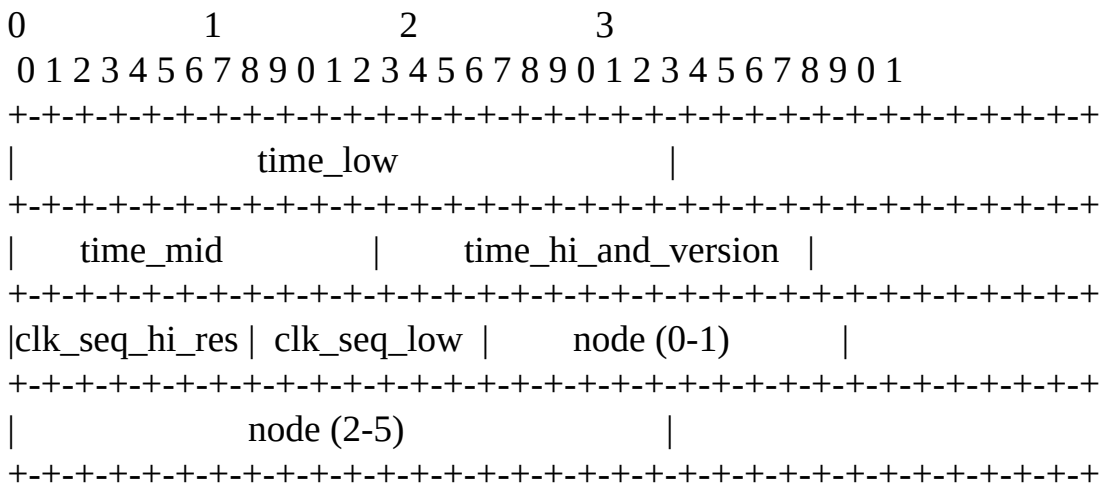
3.7 Comparing UUIDs for relative order

Two UUIDs allocated according to the same variant can also be ordered lexicographically. For the UUID variant herein defined, the first of two UUIDs follows the second if the most significant field in which the UUIDs differ is greater for the first UUID. The first of a pair of UUIDs precedes the second if the most significant field in which the UUIDs differ is greater for the second UUID.

3.8 Byte order of UUIDs

UUIDs may be transmitted in many different forms, some of which may be dependent on the presentation or application protocol where the UUID may be used. In such cases, the order, sizes and byte orders of the UUIDs fields on the wire will depend on the relevant presentation or application protocol. However, it is strongly RECOMMENDED that the order of the fields conform with ordering set out in section 3.1 above. Furthermore, the payload size of each field in the application or presentation protocol MUST be large enough that no information lost in the process of encoding them for transmission.

In the absence of explicit application or presentation protocol specification to the contrary, a UUID is encoded as a 128-bit object,



4. Node IDs when no IEEE 802 network card is available

If a system wants to generate UUIDs but has no IEE 802 compliant network card or other source of IEEE 802 addresses, then this section describes how to generate one.

The ideal solution is to obtain a 47 bit cryptographic quality random number, and use it as the low 47 bits of the node ID, with the most significant bit of the first octet of the node ID set to 1. This bit is the unicast/multicast bit, which will never be set in IEEE 802 addresses obtained from network cards; hence, there can never be a conflict between UUIDs generated by machines with and without network cards.

If a system does not have a primitive to generate cryptographic quality random numbers, then in most systems there are usually a

fairly large number of sources of randomness available from which one can be generated. Such sources are system specific, but often include:

- the percent of memory in use
- the size of main memory in bytes
- the amount of free main memory in bytes
- the size of the paging or swap file in bytes
- free bytes of paging or swap file
- the total size of user virtual address space in bytes
- the total available user address space bytes
- the size of boot disk drive in bytes
- the free disk space on boot drive in bytes
- the current time
- the amount of time since the system booted
- the individual sizes of files in various system directories
- the creation, last read, and modification times of files in various system directories
- the utilization factors of various system resources (heap, etc.)
- current mouse cursor position
- current caret position
- current number of running processes, threads
- handles or IDs of the desktop window and the active window
- the value of stack pointer of the caller
- the process and thread ID of caller
- various processor architecture specific performance counters (instructions executed, cache misses, TLB misses)

(Note that it precisely the above kinds of sources of randomness that are used to seed cryptographic quality random number generators on systems without special hardware for their construction.)

In addition, items such as the computer's name and the name of the operating system, while not strictly speaking random, will help differentiate the results from those obtained by other systems.

The exact algorithm to generate a node ID using these data is system specific, because both the data available and the functions to obtain

them are often very system specific. However, assuming that one can concatenate all the values from the randomness sources into a buffer, and that a cryptographic hash function such as MD5 [3] is available, then any 6 bytes of the MD5 hash of the buffer, with the multicast bit (the high bit of the first byte) set will be an appropriately random node ID.

Other hash functions, such as SHA-1 [4], can also be used. The only requirement is that the result be suitably random _ in the sense that the outputs from a set uniformly distributed inputs are themselves uniformly distributed, and that a single bit change in the input can be expected to cause half of the output bits to change.

5. Obtaining IEEE 802 addresses

At the time of writing, the following URL

<http://standards.ieee.org/db/oui/forms/>

contains information on how to obtain an IEEE 802 address block. At the time of writing, the cost is \$1250 US.

6. Security Considerations

It should not be assumed that UUIDs are hard to guess; they should not be used as capabilities.

7. Acknowledgements

This document draws heavily on the OSF DCE specification for UUIDs. Ted Ts'o provided helpful comments, especially on the byte ordering section which we mostly plagiarized from a proposed wording he supplied (all errors in that section are our responsibility, however).

8. References

[1] Lisa Zahn, et. al., Network Computing Architecture, Prentice Hall, Englewood Cliffs, NJ, 1990

[2] DCE: Remote Procedure Call, Open Group CAE Specification C309 ISBN 1-85912-041-5 28cm. 674p. pbk. 1,655g. 8/94

[3] R. Rivest, RFC 1321, "The MD5 Message-Digest Algorithm",
04/16/1992.

[4] NIST FIPS PUB 180-1, "Secure Hash Standard," National Institute
of Standards and Technology, U.S. Department of Commerce, DRAFT, May
31, 1994.

Leach, Salz expires Aug 1998 [Page 15]

Internet-Draft UUIDs and GUIDs (DRAFT) 02/04/98

9. Authors' addresses

Paul J. Leach
Microsoft
1 Microsoft Way
Redmond, WA, 98052, U.S.A.
paulle@microsoft.com
Tel. 425 882 8080
Fax. 425 936 7329

Rich Salz
100 Cambridge Park Drive
Cambridge MA 02140
salzr@certco.com
Tel. 617 499 4075
Fax. 617 576 0019

10. Notice

The IETF takes no position regarding the validity or scope of any
intellectual property or other rights that might be claimed to
pertain to the implementation or use of the technology described in
this document or the extent to which any license under such rights
might or might not be available; neither does it represent that it
has made any effort to identify any such rights. Information on the
IETF's procedures with respect to rights in standards-track and
standards-related documentation can be found in BCP-11. Copies of
claims of rights made available for publication and any assurances of
licenses to be made available, or the result of an attempt made to

obtain a general license or permission for the use of such proprietary rights by implementors or users of this specification can be obtained from the IETF Secretariat.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the IETF Executive Director.

11. Full Copyright Statement

Copyright (C) The Internet Society 1997. All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of

Leach, Salz expires Aug 1998 [Page 16]

Internet-Draft UUIDs and GUIDs (DRAFT) 02/04/98

developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Appendix A _ UUID Sample Implementation

This implementation consists of 5 files: uuid.h, uuid.c, sysdep.h, sysdep.c and utest.c. The uuid.* files are the system independent implementation of the UUID generation algorithms described above, with all the optimizations described above except efficient state sharing across processes included. The code has been tested on Linux (Red Hat 4.0) with GCC (2.7.2), and Windows NT 4.0 with VC++ 5.0. The code assumes 64 bit integer support, which makes it a lot clearer.

All the following source files should be considered to have the following copyright notice included:

copyrt.h

```
/*
** Copyright (c) 1990- 1993, 1996 Open Software Foundation, Inc.
** Copyright (c) 1989 by Hewlett-Packard Company, Palo Alto, Ca. &
** Digital Equipment Corporation, Maynard, Mass.
** Copyright (c) 1998 Microsoft.
** To anyone who acknowledges that this file is provided "AS IS"
** without any express or implied warranty: permission to use, copy,
** modify, and distribute this file for any purpose is hereby
** granted without fee, provided that the above copyright notices and
** this notice appears in all source code copies, and that none of
** the names of Open Software Foundation, Inc., Hewlett-Packard
** Company, or Digital Equipment Corporation be used in advertising
** or publicity pertaining to distribution of the software without
** specific, written prior permission. Neither Open Software
** Foundation, Inc., Hewlett-Packard Company, Microsoft, nor Digital
Equipment
** Corporation makes any representations about the suitability of
** this software for any purpose.
*/
```

uuid.h

Leach, Salz

expires Aug 1998

[Page 17]

Internet-Draft

UUIDs and GUIDs (DRAFT)

02/04/98

```
#include "copyrt.h"
#undef uuid_t
typedef struct _uuid_t {
    unsigned32    time_low;
    unsigned16    time_mid;
    unsigned16    time_hi_and_version;
    unsigned8     clock_seq_hi_and_reserved;
    unsigned8     clock_seq_low;
    byte          node[6];
} uuid_t;

/* uuid_create -- generate a UUID */
int uuid_create(uuid_t * uuid);

/* uuid_create_from_name -- create a UUID using a "name"
   from a "name space" */
void uuid_create_from_name(
    uuid_t * uuid,      /* resulting UUID */
    uuid_t nsid,        /* UUID to serve as context, so identical
                        names from different name spaces generate
                        different UUIDs */
    void * name,        /* the name from which to generate a UUID */
    int namelen         /* the length of the name */
);

/* uuid_compare -- Compare two UUID's "lexically" and return
   -1  u1 is lexically before u2
   0   u1 is equal to u2
   1   u1 is lexically after u2
   Note:  lexical ordering is not temporal ordering!
*/
int uuid_compare(uuid_t *u1, uuid_t *u2);
```

uuid.c

```
#include "copyrt.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "sysdep.h"
#include "uuid.h"
```

```
/* various forward declarations */
```

```
static int read_state(unsigned16 *clockseq, uuid_time_t *timestamp,
uuid_node_t * node);
static void write_state(unsigned16 clockseq, uuid_time_t timestamp,
uuid_node_t node);
static void format_uuid_v1(uuid_t * uuid, unsigned16 clockseq,
uuid_time_t timestamp, uuid_node_t node);
static void format_uuid_v3(uuid_t * uuid, unsigned char hash[16]);
static void get_current_time(uuid_time_t * timestamp);
static unsigned16 true_random(void);
```

Leach, Salz

expires Aug 1998

[Page 18]

Internet-Draft

UUIDs and GUIDs (DRAFT)

02/04/98

```
/* uuid_create -- generator a UUID */
int uuid_create(uuid_t * uuid) {
    uuid_time_t timestamp, last_time;
    unsigned16 clockseq;
    uuid_node_t node;
    uuid_node_t last_node;
    int f;

    /* acquire system wide lock so we're alone */
    LOCK;

    /* get current time */
    get_current_time(&timestamp);

    /* get node ID */
    get_ieee_node_identifier(&node);

    /* get saved state from NV storage */
    f = read_state(&clockseq, &last_time, &last_node);

    /* if no NV state, or if clock went backwards, or node ID changed
    (e.g., net card swap) change clockseq */
    if (!f || memcmp(&node, &last_node, sizeof(uuid_node_t)))
        clockseq = true_random();
    else if (timestamp < last_time)
        clockseq++;

    /* stuff fields into the UUID */
```

```
format_uuid_v1(uuid, clockseq, timestamp, node);

/* save the state for next time */
write_state(clockseq, timestamp, node);

UNLOCK;
return(1);
};

/* format_uuid_v1 -- make a UUID from the timestamp, clockseq,
    and node ID */
void format_uuid_v1(uuid_t * uuid, unsigned16 clock_seq, uuid_time_t
timestamp, uuid_node_t node) {
    /* Construct a version 1 uuid with the information we've gathered
    * plus a few constants. */
    uuid->time_low = (unsigned long)(timestamp & 0xFFFFFFFF);
    uuid->time_mid = (unsigned short)((timestamp >> 32) & 0xFFFF);
    uuid->time_hi_and_version = (unsigned short)((timestamp >> 48) &
    0x0FFF);
    uuid->time_hi_and_version |= (1 << 12);
    uuid->clock_seq_low = clock_seq & 0xFF;
    uuid->clock_seq_hi_and_reserved = (clock_seq & 0x3F00) >> 8;
    uuid->clock_seq_hi_and_reserved |= 0x80;
    memcpy(&uuid->node, &node, sizeof uuid->node);
};
```

Leach, Salz expires Aug 1998 [Page 19]

Internet-Draft UUIDs and GUIDs (DRAFT) 02/04/98

```
/* data type for UUID generator persistent state */
typedef struct {
    uuid_time_t ts;        /* saved timestamp */
    uuid_node_t node;     /* saved node ID */
    unsigned16 cs;        /* saved clock sequence */
} uuid_state;

static uuid_state st;

/* read_state -- read UUID generator state from non-volatile store */
int read_state(unsigned16 *clockseq, uuid_time_t *timestamp,
uuid_node_t *node) {
```

```
FILE * fd;
static int initied = 0;

/* only need to read state once per boot */
if (!initied) {
    fd = fopen("state", "rb");
    if (!fd)
        return (0);
    fread(&st, sizeof(uuid_state), 1, fd);
    fclose(fd);
    initied = 1;
};
*clockseq = st.cs;
*timestamp = st.ts;
*node = st.node;
return(1);
};

/* write_state -- save UUID generator state back to non-volatile
storage */
void write_state(unsigned16 clockseq, uuid_time_t timestamp,
uuid_node_t node) {
    FILE * fd;
    static int initied = 0;
    static uuid_time_t next_save;

    if (!initied) {
        next_save = timestamp;
        initied = 1;
    };
    /* always save state to volatile shared state */
    st.cs = clockseq;
    st.ts = timestamp;
    st.node = node;
    if (timestamp >= next_save) {
        fd = fopen("state", "wb");
        fwrite(&st, sizeof(uuid_state), 1, fd);
        fclose(fd);
        /* schedule next save for 10 seconds from now */
        next_save = timestamp + (10 * 10 * 1000 * 1000);
    };
};
```

```
/* get-current_time -- get time as 60 bit 100ns ticks since whenever.  
   Compensate for the fact that real clock resolution is  
   less than 100ns. */
```

```
void get_current_time(uuid_time_t * timestamp) {  
    uuid_time_t      time_now;  
    static uuid_time_t time_last;  
    static unsigned16 uuids_this_tick;  
    static int        initd = 0;  
  
    if (!initd) {  
        get_system_time(&time_now);  
        uuids_this_tick = UUIDS_PER_TICK;  
        initd = 1;  
    }  
  
    while (1) {  
        get_system_time(&time_now);  
  
        /* if clock reading changed since last UUID generated... */  
        if (time_last != time_now) {  
            /* reset count of uuids gen'd with this clock reading */  
            uuids_this_tick = 0;  
            break;  
        }  
        if (uuids_this_tick < UUIDS_PER_TICK) {  
            uuids_this_tick++;  
            break;  
        }  
        /* going too fast for our clock; spin */  
    }  
    /* add the count of uuids to low order bits of the clock reading */  
    *timestamp = time_now + uuids_this_tick;  
};
```

```
/* true_random -- generate a crypto-quality random number.  
   This sample doesn't do that. */
```

```
static unsigned16  
true_random(void)  
{  
    static int initd = 0;  
    uuid_time_t time_now;
```



```
if (!inited) {
    get_system_time(&time_now);
    time_now = time_now/UUIDS_PER_TICK;
    srand(((unsigned int)(((time_now >> 32) ^ time_now)&0xffffffff)));
    inited = 1;
};

return (rand());
}
```

Leach, Salz expires Aug 1998 [Page 21]

Internet-Draft UUIDs and GUIDs (DRAFT) 02/04/98

```
/* uuid_create_from_name -- create a UUID using a "name" from a "name
space" */
void uuid_create_from_name(
    uuid_t * uuid,              /* resulting UUID */
    uuid_t nsid,              /* UUID to serve as context, so identical
names from different name spaces generate
different UUIDs */
    void * name,              /* the name from which to generate a UUID */
    int namelen              /* the length of the name */
) {
    MD5_CTX c;
    unsigned char hash[16];
    uuid_t net_nsid;              /* context UUID in network byte order */

    /* put name space ID in network byte order so it hashes the same
no matter what endian machine we're on */
    net_nsid = nsid;
    htonl(net_nsid.time_low);
    htons(net_nsid.time_mid);
    htons(net_nsid.time_hi_and_version);

    MD5Init(&c);
    MD5Update(&c, &net_nsid, sizeof(uuid_t));
    MD5Update(&c, name, namelen);
    MD5Final(hash, &c);
```

```
/* the hash is in network byte order at this point */
format_uuid_v3(uuid, hash);
};

/* format_uuid_v3 -- make a UUID from a (pseudo)random 128 bit number
*/
void format_uuid_v3(uuid_t * uuid, unsigned char hash[16]) {
    /* Construct a version 3 uuid with the (pseudo-)random number
    * plus a few constants. */

    memcpy(uuid, hash, sizeof(uuid_t));

    /* convert UUID to local byte order */
    ntohl(uuid->time_low);
    ntohs(uuid->time_mid);
    ntohs(uuid->time_hi_and_version);

    /* put in the variant and version bits */
    uuid->time_hi_and_version &= 0x0FFF;
    uuid->time_hi_and_version |= (3 << 12);
    uuid->clock_seq_hi_and_reserved &= 0x3F;
    uuid->clock_seq_hi_and_reserved |= 0x80;
};

/* uuid_compare -- Compare two UUID's "lexically" and return
    -1  u1 is lexically before u2
    0   u1 is equal to u2
    1   u1 is lexically after u2
```

Leach, Salz

expires Aug 1998

[Page 22]

Internet-Draft

UUIDs and GUIDs (DRAFT)

02/04/98

Note: lexical ordering is not temporal ordering!

```
*/
int uuid_compare(uuid_t *u1, uuid_t *u2)
{
    int i;

#define CHECK(f1, f2) if (f1 != f2) return f1 < f2 ? -1 : 1;
    CHECK(u1->time_low, u2->time_low);
    CHECK(u1->time_mid, u2->time_mid);
    CHECK(u1->time_hi_and_version, u2->time_hi_and_version);
```

```
CHECK(u1->clock_seq_hi_and_reserved, u2->clock_seq_hi_and_reserved);
CHECK(u1->clock_seq_low, u2->clock_seq_low)
for (i = 0; i < 6; i++) {
    if (u1->node[i] < u2->node[i])
        return -1;
    if (u1->node[i] > u2->node[i])
        return 1;
}
return 0;
};
```

sysdep.h

```
#include "copyrt.h"
/* remove the following define if you aren't running WIN32 */
#define WININC 0

#ifdef WININC
#include <windows.h>
#else
#include <sys/types.h>
#include <sys/time.h>
#include <sys/sysinfo.h>
#endif

/* change to point to where MD5 .h's live */
/* get MD5 sample implementation from RFC 1321 */
#include "global.h"
#include "md5.h"

/* set the following to the number of 100ns ticks of the actual
resolution of
your system's clock */
#define UUIDS_PER_TICK 1024

/* Set the following to a call to acquire a system wide global lock
*/
#define LOCK
#define UNLOCK

typedef unsigned long    unsigned32;
typedef unsigned short   unsigned16;
typedef unsigned char     unsigned8;
typedef unsigned char    byte;
```

Internet-Draft

UUIDs and GUIDs (DRAFT)

02/04/98

```
/* Set this to what your compiler uses for 64 bit data type */
#ifdef WININC
#define unsigned64_t unsigned __int64
#define I64(C) C
#else
#define unsigned64_t unsigned long long
#define I64(C) C##LL
#endif
```

```
typedef unsigned64_t uuid_time_t;
typedef struct {
    char nodeID[6];
} uuid_node_t;
```

```
void get_ieee_node_identifier(uuid_node_t *node);
void get_system_time(uuid_time_t *uuid_time);
void get_random_info(char seed[16]);
```

sysdep.c

```
#include "copyrt.h"
#include <stdio.h>
#include "sysdep.h"
```

```
/* system dependent call to get IEEE node ID.
   This sample implementation generates a random node ID
   */
```

```
void get_ieee_node_identifier(uuid_node_t *node) {
    char seed[16];
    FILE * fd;
    static initd = 0;
    static uuid_node_t saved_node;

    if (!initd) {
        fd = fopen("nodeid", "rb");
        if (fd) {
            fread(&saved_node, sizeof(uuid_node_t), 1, fd);

```

```
        fclose(fd);
    }
    else {
        get_random_info(seed);
        seed[0] |= 0x80;
        memcpy(&saved_node, seed, sizeof(uuid_node_t));
        fd = fopen("nodeid", "wb");
        if (fd) {
            fwrite(&saved_node, sizeof(uuid_node_t), 1, fd);
            fclose(fd);
        };
    };
    initied = 1;
};
```

Leach, Salz

expires Aug 1998

[Page 24]

Internet-Draft

UUIDs and GUIDs (DRAFT)

02/04/98

```
*node = saved_node;
};
```

```
/* system dependent call to get the current system time.
```

```
Returned as 100ns ticks since Oct 15, 1582, but resolution may be
less than 100ns.
```

```
*/
```

```
#ifdef _WINDOWS_
```

```
void get_system_time(uuid_time_t *uuid_time) {
    ULARGE_INTEGER time;
```

```
    GetSystemTimeAsFileTime((FILETIME *)&time);
```

```
/* NT keeps time in FILETIME format which is 100ns ticks since
Jan 1, 1601. UUIDs use time in 100ns ticks since Oct 15, 1582.
The difference is 17 Days in Oct + 30 (Nov) + 31 (Dec)
+ 18 years and 5 leap days.
```

```
*/
```

```
time.QuadPart +=
```

```
    (unsigned __int64) (1000*1000*10)    // seconds
```

```
    * (unsigned __int64) (60 * 60 * 24)    // days
```

```
    * (unsigned __int64) (17+30+31+365*18+5); // # of days
```

```
*uuid_time = time.QuadPart;

};

void get_random_info(char seed[16]) {
    MD5_CTX c;
    typedef struct {
        MEMORYSTATUS m;
        SYSTEM_INFO s;
        FILETIME t;
        LARGE_INTEGER pc;
        DWORD tc;
        DWORD l;
        char hostname[MAX_COMPUTERNAME_LENGTH + 1];
    } randomness;
    randomness r;

    MD5Init(&c);
    /* memory usage stats */
    GlobalMemoryStatus(&r.m);
    /* random system stats */
    GetSystemInfo(&r.s);
    /* 100ns resolution (nominally) time of day */
    GetSystemTimeAsFileTime(&r.t);
    /* high resolution performance counter */
    QueryPerformanceCounter(&r.pc);
    /* milliseconds since last boot */
    r.tc = GetTickCount();
    r.l = MAX_COMPUTERNAME_LENGTH + 1;
```

Leach, Salz expires Aug 1998 [Page 25]

Internet-Draft UUIDs and GUIDs (DRAFT) 02/04/98

```
    GetComputerName(r.hostname, &r.l );
    MD5Update(&c, &r, sizeof(randomness));
    MD5Final(seed, &c);
};
#else
```

```
void get_system_time(uuid_time_t *uuid_time)
{
```

```
struct timeval tp;

gettimeofday(&tp, (struct timezone *)0);

/* Offset between UUID formatted times and Unix formatted times.
   UUID UTC base time is October 15, 1582.
   Unix base time is January 1, 1970.
*/
*uuid_time = (tp.tv_sec * 100000000) + (tp.tv_usec * 10) +
    I64(0x01B21DD213814000);
};

void get_random_info(char seed[16]) {
    MD5_CTX c;
    typedef struct {
        struct sysinfo s;
        struct timeval t;
        char hostname[257];
    } randomness;
    randomness r;

    MD5Init(&c);
    sysinfo(&r.s);
    gettimeofday(&r.t, (struct timezone *)0);
    gethostname(r.hostname, 256);
    MD5Update(&c, &r, sizeof(randomness));
    MD5Final(seed, &c);
};

#endif

utest.c

#include "copyrt.h"
#include "sysdep.h"
#include <stdio.h>
#include "uuid.h"

uuid_t Namespace_DNS = { /* 6ba7b810-9dad-11d1-80b4-00c04fd430c8 */
    0x6ba7b810,
    0x9dad,
    0x11d1,
    0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8
};
```

Leach, Salz

expires Aug 1998

[Page 26]

Internet-Draft

UUIDs and GUIDs (DRAFT)

02/04/98

```
/* puid -- print a UUID */
void puid(uuid_t u);

/* Simple driver for UUID generator */
void main(int argc, char **argv) {
    uuid_t u;
    int f;

    uuid_create(&u);
    printf("uuid_create()      -> "); puid(u);

    f = uuid_compare(&u, &u);
    printf("uuid_compare(u,u): %d\n", f); /* should be 0 */
    f = uuid_compare(&u, &NameSpace_DNS);
    printf("uuid_compare(u, NameSpace_DNS): %d\n", f); /* s.b. 1 */
    f = uuid_compare(&NameSpace_DNS, &u);
    printf("uuid_compare(NameSpace_DNS, u): %d\n", f); /* s.b. -1 */

    uuid_create_from_name(&u, NameSpace_DNS, "www.widgets.com", 15);
    printf("uuid_create_from_name() -> "); puid(u);
};

void puid(uuid_t u) {
    int i;

    printf("%8.8x-%4.4x-%4.4x-%2.2x%2.2x-", u.time_low, u.time_mid,
        u.time_hi_and_version, u.clock_seq_hi_and_reserved,
        u.clock_seq_low);
    for (i = 0; i < 6; i++)
        printf("%2.2x", u.node[i]);
    printf("\n");
};
```

Appendix B _ Sample output of utest

```
uuid_create()      -> 7d444840-9dc0-11d1-b245-5ffdce74fad2
uuid_compare(u,u): 0
uuid_compare(u, NameSpace_DNS): 1
```



```
uuid_compare(NameSpace_DNS, u): -1
uuid_create_from_name() -> e902893a-9d22-3c7e-a7b8-d6e313b71d9f
```

Appendix C _ Some name space IDs

This appendix lists the name space IDs for some potentially interesting name spaces, as initialized C structures and in the string representation defined in section 3.5

```
uuid_t NameSpace_DNS = { /* 6ba7b810-9dad-11d1-80b4-00c04fd430c8 */
    0x6ba7b810,
    0x9dad,
    0x11d1,
    0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8
};
```

Leach, Salz expires Aug 1998 [Page 27]

Internet-Draft UUIDs and GUIDs (DRAFT) 02/04/98

```
uuid_t NameSpace_URL = { /* 6ba7b811-9dad-11d1-80b4-00c04fd430c8 */
    0x6ba7b811,
    0x9dad,
    0x11d1,
    0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8
};
```

```
uuid_t NameSpace_OID = { /* 6ba7b812-9dad-11d1-80b4-00c04fd430c8 */
    0x6ba7b812,
    0x9dad,
    0x11d1,
    0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8
};
```

```
uuid_t NameSpace_X500 = { /* 6ba7b814-9dad-11d1-80b4-00c04fd430c8 */
    0x6ba7b814,
    0x9dad,
    0x11d1,
    0x80, 0xb4, 0x00, 0xc0, 0x4f, 0xd4, 0x30, 0xc8
};
```




[Home](#) | [Site map](#) | [Abbreviations](#) | [ISO Store](#)
[Français](#) | [FAQ](#) | [Contact ISO](#) | [My account](#)
Search Extended Search

[About ISO](#) | [Products and services](#) | [ISO 9000 / 14000](#) | [Standards development](#) | [Communities and markets](#) | [Communication centre](#)



ISO Catalogue

ICS fields

35

Information technology.
Office machines

35.060

Languages used in information technology



[View Shopping Basket](#)

The URL you've requested doesn't link to a valid catalogue entry.

Search options

- ☒ Text
☐ ISO Number

Type in search string

[Help on using search](#)

[Extended Search](#)

- [How to use the catalogue](#)
- [Maintenance agencies and Registration Authorities](#)
- [List of withdrawn standards](#)

© ISO | [ISO name and logo](#) | [Privacy policy](#)

▲ Vorlesung Datenbanken

▼ 1 Motivation und Einführung

▼ 1.1 Begriffsbestimmung: Was ist eine Datenbank?

▼ 1.2 Anforderungen an Datenbanksysteme

▼ 1.3 Typen von Datenbankmanagementsystemen

▼ 2 Entwurf einer Datenbank

▼ 2.1 Graphischer Entwurf des konzeptuellen Schemas mit dem Entity-Relationship Modell

▼ 2.2 Ableitung logischer Relationenstrukturen

▼ 2.3 Algebraischer Entwurf mit der Normalformentheorie

▼ 3 Arbeiten mit einer Datenbank

▼ 3.1 Codd'sche Regeln und Eigenschaften relationaler Systeme

▼ 3.2 Implementierung des logischen Modells mit SQL-DDL

▼ 3.3 Der Anfrageteil von SQL

▼ 3.4 Der Datenmanipulationsteil von SQL

▶ Empfohlene Literatur

▲ Hinweis

Aufgrund der verfügbaren Menge guter einführender (auch deutschsprachiger) Datenbankliteratur verzichtet das vorliegende Scriptum darauf den in der Literatur verfügbaren Stoff nochmals aufzubereiten, sondern versammelt die Definitionen, Beispiele und Anmerkungen der Vorlesungen in einer übersichtlichen Zusammenstellung.

Für die vertiefende ausführliche lehrbuchartige Darstellung des Stoffes sei auf die [empfohlene Literatur](#) verwiesen.

▲ 1 Motivation und Einführung

1.1 Begriffsbestimmung: Was ist eine Datenbank?

Motivation für die Einführung einer Datenbank anstatt selbsterstellter Verwaltungs- und Zugriffsroutinen:

- Daten-Programm-Unabhängigkeit.
Die verwalteten Daten sollen unabhängig vom sie verarbeiteten Programm gespeichert und zugreifbar sein.
Dies wäre zwar in einem ersten Schritt auch durch die Verwendung des Dateisystems möglich, allerdings würde hierfür ein Programm zur Abbildung der Programmdaten auf die Dateistrukturen benötigt, welches selbst wieder eine Abhängigkeitsbeziehung zwischen Daten und Programm --- nun eben dem Abbildungsprogramm --- darstellen würde.
- Flexible Speicherung.
Datenbankmanagementsysteme speichern die verwalteten Daten deutlich flexibler als selbsterstellte Routinen und sind somit hinsichtlich der Zukunftsfähigkeit effizienter.
- Verwaltungsfunktionen.
Datenbankmanagementsysteme bieten in der Regel eine Reihe über die reine Datenverwaltung hinausgehende Funktionen wie Backup-Recovery, Integritätssicherung, Synchronisation gleichzeitiger Zugriffe oder Transaktionskontrolle an, die nicht selbständig implementiert werden müssen.

Grundlegende Begriffe

Definition 1: Daten

Daten sind durch die Maschine verarbeitbare Einheiten.

Definition 2: Information

Daten die Bedeutung für den Empfänger besitzen.

Nach Shannon ermißt sich der Wert einer Information durch den Zuwachs der durch den Adressaten nach Kenntnis der Information beantwortbaren Ja/Nein-Fragen.

Mehr zum Unterschied zwischen *Daten* und *Information*:

- [Homepage des Arbeitskreises Bildung, einem Zusammenschluß von Stipendiaten der Friedrich Naumann Stiftung](#)
- [Glossar der deutschsprachigen Anleitung zu PGP](#)

Definition 3: Datenbank

Eine Datenbank (engl. *data base*) ist ein integrierter, persistenter Datenbestand einschließlich aller relevanten Informationen über die dargestellten Information (sog. Metainformation, d.h. Integritätsbedingungen und Regeln), der einer Gruppe von Benutzern in nur einem Exemplar zur Verfügung steht und durch ein [DBMS](#) verwaltet wird.

Definition 4: Datenbankmanagementsystem (DBMS)

Ein Datenbankmanagementsystem (DBMS) ist die Gesamtheit aller Programme zur Erzeugung, Verwaltung und Manipulation einer [Datenbank](#).

Im Deutschen wird auch der Begriff *Datenbankverwaltungssystem* (DBVS) synonym verwendet.

Beispiele verfügbarer DBMS:

- Die DBMS-Produkte des Herstellers [Sybase](#)
- [IDMS](#) von [Computer Associates](#)
- [IMS](#), [DB2](#) und [Informix](#) von [IBM](#)
- [MySQL](#) des gleichnamigen Herstellers
- [Oracle 9i](#) des Herstellers [Oracle](#)
- [SQLServer](#) und [Access](#) des Herstellers [Microsoft](#)

Beispiel 1: Am Markt verfügbare DBM-Systeme**Definition 5: Relationales DBMS**

Ein relationales Datenbankmanagementsystem (RDBMS) ist ein [DBMS](#), welches intern gemäß dem [relationalen Modell](#) organisiert ist.

Bei den genannten DBMS *MySQL*, *SQLServer*, *Access*, *DB2* und *Oracle* handelt es sich um relationale Systeme, bzw. Weiterentwicklungen davon.

Beispiel 2: Am Markt verfügbare RDBM-Systeme**Definition 6: Relation**

Eine Relation $R(A_1, A_2 \dots A_n)$ ist eine benannte Menge von n -Tupeln, wobei ein n -Tupel eine Anordnung von n atomaren, d.h. einfachen (nicht weiter zerlegbaren) Attributen $A_1, A_2 \dots A_n$ ist.

Die Relation *Person* mit den Attributen *Vorname*, *Nachname* und *Geburtsdatum*.

Werteausprägungen davon:

Person₁("Meier", "Schorsch", "1955-10-01")

Person₂("Huber", "Franz", "1945-08-03")

...

Die Relation *Student* mit den Attributen *Name*, *Matrikelnummer*, *Semester* und *regelmäßigerMensabesucher*.

Werteausprägungen davon:

Student₁("Meier Schorsch", "08154711", "WIB 1", "true")

Student₂("Müller Xaver", "73619452", "BCM 4", "false")

...

Beispiel 3: Relationen

Definition 7: Tabelle

Eine Tabelle unterscheidet sich von einer [Relation](#) darin, daß ein Tupel mehrfach auftreten darf; eine Tabelle ist mathematisch keine Menge.

Die Tabelle *Student* mit den Attributen *Name*, *Matrikelnummer*, *Semester* und *regelmäßigerMensabesucher*.

Werteausprägungen davon:

Student₁("Meier Schorsch", "08154711", "WIB 1", "true")

Student₂("Müller Xaver", "73619452", "BCM 4", "false")

Student₃("Müller Xaver", "73619452", "BCM 4", "false")

...

Man beachte, daß der dritte Eintrag doppelt vorkommt, d.h. in all seinen Wertbelegungen mit dem zweiten übereinstimmt.

Beispiel 4: Tabelle**Definition 8: Modell**

Ein Modell bildet einen existierenden Sachverhalt deskriptiv nach oder nimmt einen Zukünftigen präskriptiv voraus.

Teilweise wird der Begriff *Schema* synonym gebraucht.

Deskriptive Modelle: Modelleisenbahn, Stadtplan, Photo.

Präskriptive Modelle: Bauplan eines Hauses, Skizze eines Gemäldes, maßstäblich verkleinerte Skulptur als Vorbild.

Beispiel 5: Modelle**Definition 9: Datenbanksprache**

Eine Sprache die zur Erzeugung oder Interaktion mit den Daten bzw. zu deren Verwaltung eingesetzt wird.

Es werden unterschieden:

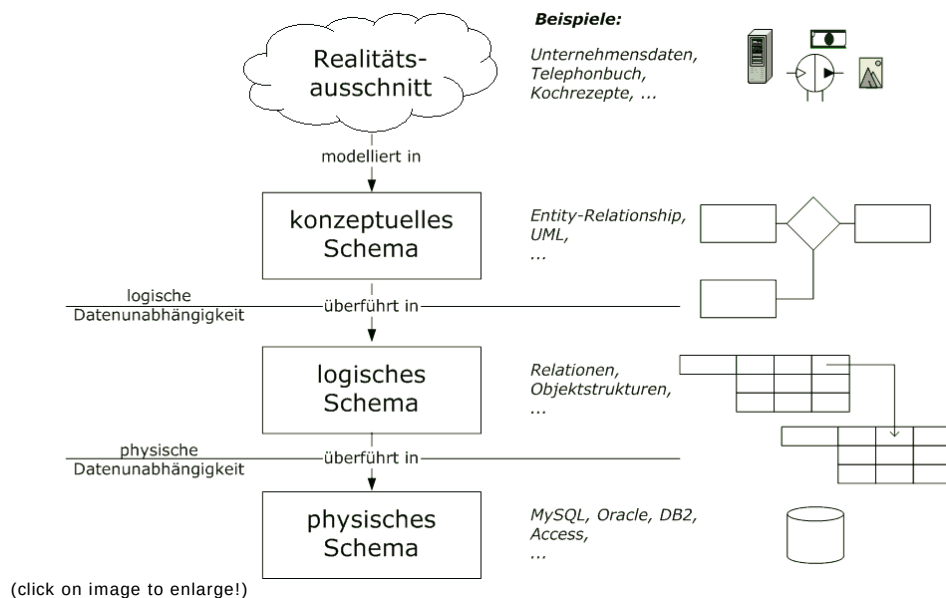
- Data Definition Language (DDL).
Zur Erzeugung eines Datenmodells.
- Data Manipulation Language (DML).
Zur Modifikation der verwalteten Daten.
- Data Retrieval Language (DRL).
Zur Anfrage der in einer [Datenbank](#) gespeicherten Daten.
- Data Control Language (DCL).
Zur Festlegung und Kontrolle von Zugriffsberechtigungen.

Im Verlauf der Vorlesung wird mit SQL die bekannteste Sprache im Umfeld relationaler DBMS eingeführt.

Beispiel einer SQL-Anfrage:

```
SELECT FNAME, BDATE FROM EMPLOYEE ORDERED BY BDATE
```

Beispiel 6: Die Datenbanksprache SQL**3-Schema-Architektur****Abbildung 1: 3-Schema-Architektur**



Die [Abbildung 1](#) stellt die 3-Schema-Architektur dar, welche die drei zentralen Modelltypen des Datenbankentwurfsprozesses miteinander in Beziehung setzt.

Definition 10: Konzeptuelles Schema

Ein konzeptuelles Schema ist ein [Modell](#), welches den relevanten Realitätsausschnitt (auch *Miniwelt*, *Diskursbereich* oder *Universe of Discourse* genannt) in Struktur und Inhalt beschreibt.

Definition 11: Logisches Schema

Ein logisches Schema ist ein [Modell](#), welches paradigmenspezifisch aus einem [konzeptuellen Schema](#) abgeleitet wurde.

Die Definition von Relationen als mathematisches Konzept zur Datenstrukturierung stellt ein logisches Schema dar.
Beispielsweise die Festlegung der Struktur der *Person* oder des *Studenten* in [Beispiel 3](#).

Beispiel 7: Relationen sind ein logisches Schema

Definition 12: Physisches Schema

Ein physisches Schema ist ein implementierungsspezifisches [Modell](#), welches aus einem [logischen Schema](#) abgeleitet wurde.

Definition 13: Datenunabhängigkeit

Die Formulierung einer Modellschicht (d.h. eines Datenmodells) ist von den darunter- bzw. darüberliegenden Modellschichten dann datenunabhängig, wenn Änderungen in den „umgebenden“ Modellschichten sich nicht auf die betrachtete Modellschicht auswirken.

Der Vorgang der *Ableitung* zwischen den verschiedenen Modelltypen der 3-Schema-Architektur sollte hierbei idealerweise (aus Gründen der Überprüfbarkeit, Nachvollziehbarkeit, Wiederholbarkeit und Qualitätssicherung) durch einen deterministischen Algorithmus erfolgen.

Die [Abbildung 1](#) zeigt rechts neben den Modelltypen symbolhaft typische graphische Veranschaulichungen der jeweiligen Modellausprägungen.

1.2 Anforderungen an Datenbanksysteme

Allgemein: Speicherung, Verwaltung und Kontrolle der Daten sowie Organisation des u.U. gleichzeitig erfolgenden Zugriffs.

Spezieller:

- Redundanzfreie Datenspeicherung.
Von dieser Forderung kann bewußt aus Gründen der Geschwindigkeitsoptimierung abgewichen werden.
- Gewährleistung von Integritätsbedingungen und Einhaltung von Regeln.

- Daten-Programm-Unabhängigkeit.

Wünschenswerte Eigenschaften:

- Leistungsfähigkeit
- Skalierbarkeit
- Benutzerfreundlichkeit
- Flexibilität
- ... spezifische Anforderungen, die sich aus der Anwendungssituation ergeben

1.3 Typen von Datenbankmanagementsystemen

Datenbanken werden heute vielfältig in Wirtschaft, Technik und Wissenschaft eingesetzt. Für verschiedene Anwendungsgebiete und Strukturen der verwalteten Daten haben sich daher spezifische DBMS-(Unter-)Typen herausgebildet, die diese Anwendungsfelder besonders gut unterstützen:

- **Deduktive Datenbanken**
Ein um eine Menge von Regeln (Deduktionskomponenten) erweitertes Datenmodell welches logische Schlüsse auf Basis der hinterlegten Fakten ziehen kann.
- **Multimedia Datenbanken**
Ein System, welches sich besonders zur Verwaltung großer Bild-, Audio- oder Videodaten eignet.
- **Objektdatenbanken**
Ein System zur Speicherung von Strukturen gemäß dem logischen Objektmodell.
- **Geographische Datenbanken**
Ein System das sich besonders zur Verwaltung geographischer Daten (z.B. Landkarten) eignet.
- **XML-Datenbanken**
Ein System zur Speicherung gemäß dem logischen Modell des XML Information Sets.
- **Aktive Datenbanken**
Ein System zur selbständigen Reaktion auf externe Ereignisse.
- **Temporale Datenbanken**
Ein System, welches neben den reinen Datenbeständen auch die Zeit des Datenzustandes mitverwaltet.

▲ Exkurs

[Erste Gehversuche mit dem RDBMS MySQL](#)

▲ 2 Entwurf einer Datenbank

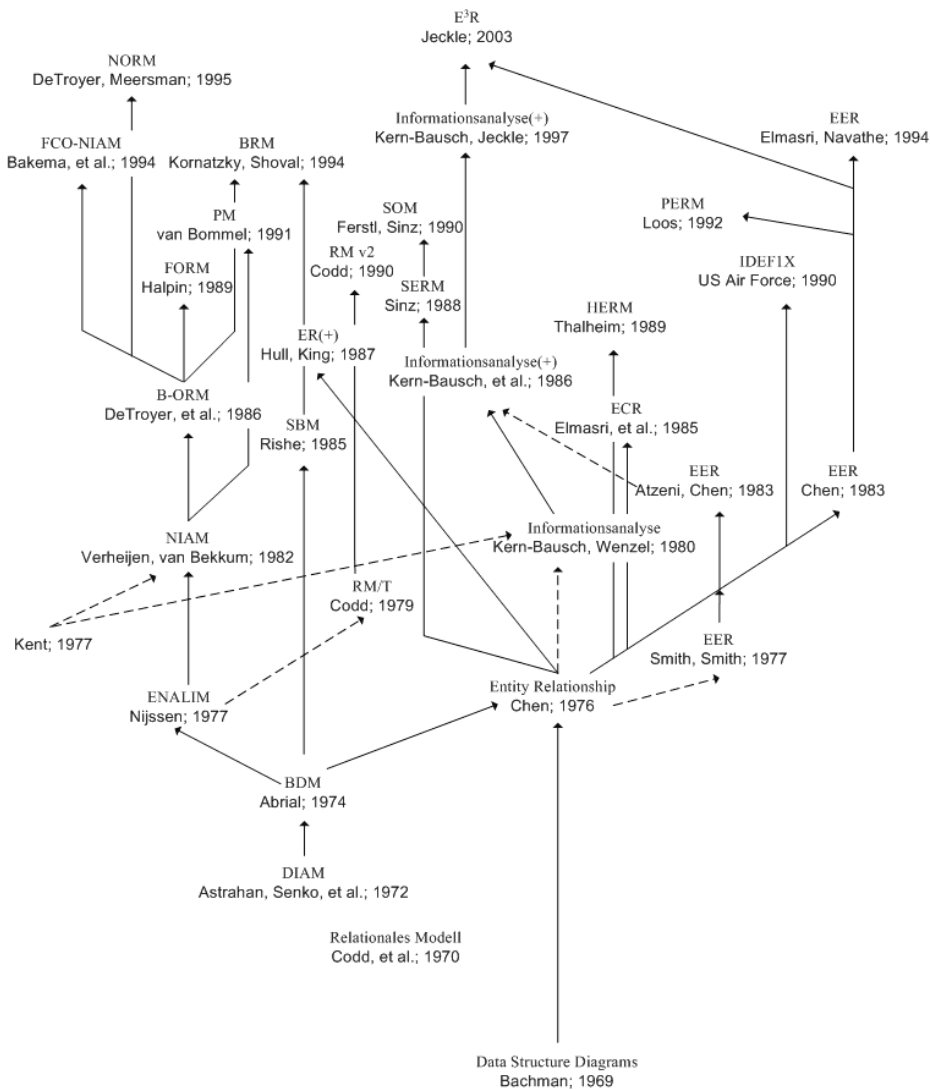
2.1 Graphischer Entwurf des konzeptuellen Schemas mit dem Entity-Relationship Modell

Seit der wirkungsmächtigen Erstveröffentlichung des *Entity Relationship Modells* (ERM) durch P. Chen 1976 kommt dieser Modellierungssprache zur Erstellung des konzeptuellen Schemas die uneingeschränkt größte Bedeutung in der Praxis zu.

In der Folgezeit wurden verschiedene Weiterentwicklungen des ursprünglichen ERM vorgeschlagen, die das Originalmodell in verschiedenen Richtungen erweitern. Hierunter fallen die Einführung von Konstrukten zur Abbildung hierarchischer Beziehungen ebenso wie Primitive zur Darstellung von Aggregationsbeziehungen.

Die Graphik der [Abbildung 2](#) zeigt eine Auswahl verschiedener Entwicklungen rund um das initiale ERM sowie einige zentrale Weiterentwicklungen. Innerhalb der Abbildung ist unterhalb des Namens der Modellierungssprache (sofern vorhanden, bei Weiterentwicklungen ohne eigenständige Namensgebung ist zur Unterscheidung vom Vorgängermodell ein geklammertes Pluszeichen angetragen) der Autor sowie das Jahr der Erstveröffentlichung dargestellt.

Abbildung 2: Entwicklungslinien des ER-Modells



(click on image to enlarge!)

Im oberen Bereich der Abbildung ist das *Semantically Enriched Extended Entity Relationship Model* (E³R) dargestellt, welches im Rahmen dieser Vorlesung behandelt wird. Es stellt eine kompatible Entwicklung dar, die versucht die datenorientierten Aspekte der ER-Nachfolgemodelle mit denen der semantischen Datenmodellierung zu vereinigen.

Die Grundkonzepte des E³R-Modells sind:

Definition 14: Entität

Eine Entität ist ein eindeutig identifizierbares und daher wohlunterscheidbares „Ding“.

Anmerkung: Der Begriff *Ding* wird hierbei in seiner Bedeutung als Synonym von *Seiendes*, *Gegenstand* oder *Objekt* gebraucht. Die philosophische Terminologie detailliert den Begriff zusätzlich hinsichtlich seiner Verwendung zur Beschreibung raumzeitlicher Gegenstände mit festgelegten charakteristischen (substantiellen) und zufällig anhaftenden Eigenschaften (Akzidenzien) aus.

- Die Tafel direkt vor ihnen.
- Sie selbst.
- Dieser Hörsaal.

Beispiel 8: Beispiele für Entitäten

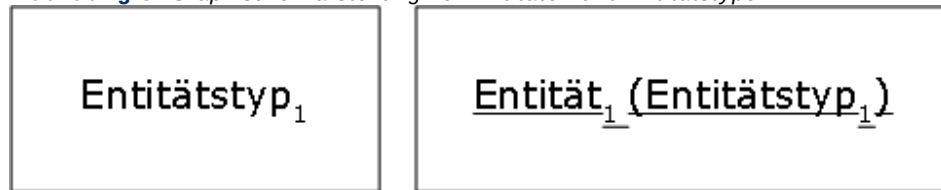
Definition 15: Entitätstyp

Ein Entitätstyp ist eine ungeordnete und duplikatfreie Sammlung von als logische zusammengehörig betrachteten [Entitäten](#).

- Tafel.
- Person.
- Student.

Beispiel 9: Beispiele für Entitätstypen

Entitätstypen werden graphisch durch benannte Rechtecke und Entitäten durch unterstrichene benannte Rechtecke dargestellt, die auf den Entitätsnamen folgend den Namen des Entitätstypen in Klammern tragen.

Abbildung 3: Graphische Darstellung von Entitäten und Entitätstypen

(click on image to enlarge!)

Soll ein Hinweis auf eine spätere physische Realisierung (d.h. die gewählte Form der Abspeicherung von Entitäten in der Datenbank) gegeben werden, so kann einem Entitätstypen ein Repräsentationstyp zugeordnet werden, bzw. einer Entität eine Repräsentation.

Definition 16: *Repräsentationstyp*

Ein Repräsentationstyp führt einen physischen Typ in das konzeptuelle Schema ein, der zur technischen Implementierung eines durch ihn annotierten Entitätstypen herangezogen werden kann.

Als Repräsentationstypen sind beliebige atomare (d.h. in ihrer Semantik nicht weiter verlustfrei zerlegbare) Datentypen eines logischen oder physischen Modells zugelassen.

- Integer
- Datum
- Money
- String

Beispiel 10: Beispiele für Repräsentationstypen**Definition 17:** *Repräsentation*

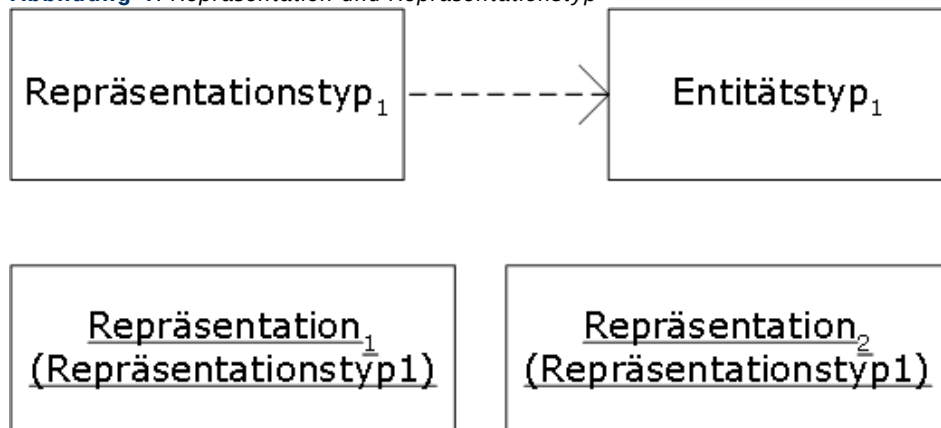
Eine Repräsentation ist eine Ausprägung genau eines Repräsentationstypen.

Der zugehörige Repräsentationstyp ist in Klammern angegeben.

- 42 (Integer)
- 2004-01-08 (Datum)
- €99,95 (Money)
- "Hallo Welt!" (String)

Beispiel 11: Beispiele für Repräsentationen

Die graphische Darstellung erfolgt durch benannte Rechtecke. Repräsentationen werden unterstrichen mit der geklammerten nachfolgenden Angabe des Repräsentationstypen dargestellt. Repräsentationstypen werden durch eine gerichtete Kante mit unterbrochener Linienführung mit dem durch sie repräsentierten Entitätstypen verbunden.

Abbildung 4: *Repräsentation und Repräsentationstyp*

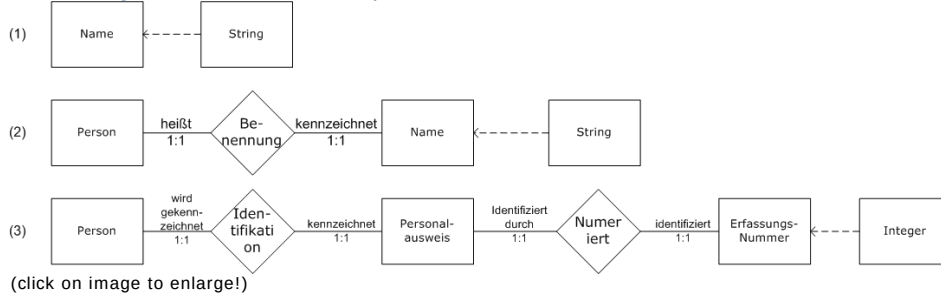
(click on image to enlarge!)

Das Beispiel der Abbildung 5 zeigt verschiedene Beispiele für die Verknüpfung von Entitätstypen mit ihren zugehörigen Repräsentationstypen.

Im Teilbeispiel (1) wird der Entitätstyp Name unmittelbar durch den Repräsentationstypen String repräsentiert, d.h. die spätere physische Realisierung des Entitätstypen Name wird durch den Datentyp String erfolgen.

Teilbeispiel (2) zeigt eine transitive Repräsentation (genaugenommen eine transitive Repräsentation erster Ordnung). Hier ist der Entitätstyp Person, welcher selbst über keinen Repräsentationstypen verfügt, in eindeutiger Weise (d.h. über einen Assoziationstyp der ausschließlich über Kardinalitätsintervalle von 1:1 verfügt (nach Maßgabe der [Anmerkung zur Struktur der Kardinalitätsintervalle](#) kann es sich daher nur um einen binären Assoziationstypen handeln)) mit dem Entitätstypen Name verknüpft, der über die Repräsentation String verfügt. Abschließend zeigt das Teilbeispiel (3) die transitive eindeutige Assoziierung des Entitätstypen Person mit dem Entitätstypen Personalausweis durch den Assoziationstypen Identifikation, wobei Personalausweis seinerseits in eindeutiger Weise mit der durch Integer repräsentierten Erfassungsnummer assoziiert ist.

Abbildung 5: Identifizierende Repräsentationen



Definition 18: Assoziation

Eine Assoziation ist eine benannte n -äre Beziehung ($n > 1$) zwischen [Entitäten](#). Die Semantik jeder durch eine Assoziation verbundenen [Entität](#) wird durch Angabe einer innerhalb einer Assoziation für jede verbundene Entität eindeutigen Rolle konkretisiert.

Definition 19: Assoziationstyp

Ein Assoziationstyp ist eine duplikatfreie ungeordnete Sammlung von logisch als zusammengehörig betrachteten [Assoziationen](#).

Jede zu einem Assoziationstypen beitragende [Rolle](#) wird durch ein [Kardinalitätsintervall](#) ergänzt.

- Arbeitsverhältnis.
- Ehe.
- Verwandtschaft.

Beispiel 12: Beispiele für Assoziationstypen

Definition 20: Kardinalitätsintervall

Ein Kardinalitätsintervall legt die Anzahl derjenigen [Entitäten](#) fest, die mit einer die [Rolle](#) einnehmenden [Entität](#) zu einem Zeitpunkt innerhalb einer [Assoziation](#) verbunden sein können. Das Intervall wird in der Schreibweise „ $i:j$ “ angegeben, wobei i eine beliebige natürliche Zahl oder die Null ist und j eine beliebige natürliche Zahl oder das Symbol n ist. Zusätzlich gilt: $i \leq j$.

- 0:1.
- 3:7.
- 0:n.
- 1:n.
- 99:n.

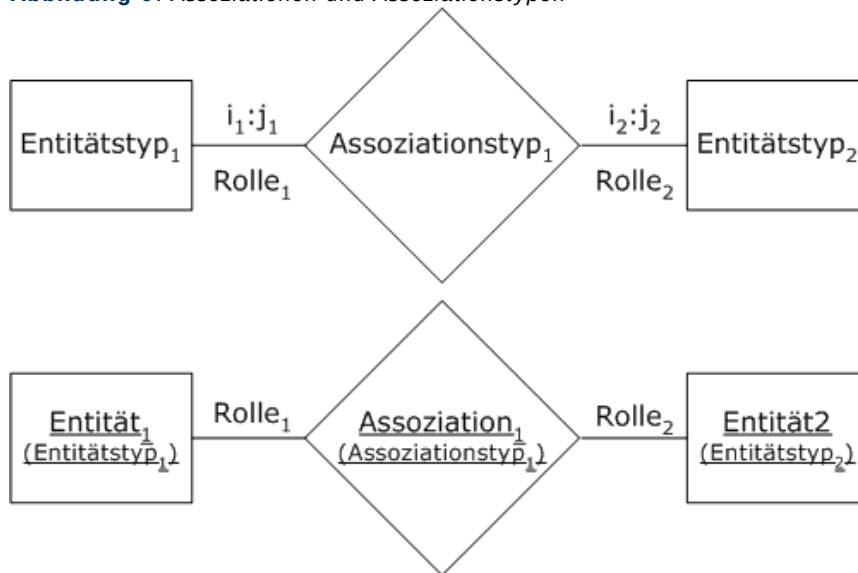
Beispiel 13: Beispiele für Kardinalitätsintervalle

Ungültig hingegen sind:

- 7:0 (Obergrenze kleiner als Untergrenze).
- -5:7 (-5 ist keine natürliche Zahl.)
- $n:8$ (n ist nicht als Untergrenze erlaubt.)

Allgemein gilt: Für n -äre [Assoziationstypen](#) gilt die Einschränkung, daß die Maximalkardinalität die ein [Entitätstyp](#) zu einem n -ären [Assoziationstypen](#) beitragen darf größer gleich $n-1$ ist.

Graphisch werden [Assoziationen](#) und [Assoziationstypen](#) durch benannte Rauten dargestellt in deren Zentrum der Name des [Assoziationstypen](#), bzw. unterstrichen der Name der Assoziation gefolgt vom Namen des Assoziationstypen, dargestellt ist.

Abbildung 6: Assoziationen und Assoziationstypen

(click on image to enlarge!)

Zentrales Konzept des E³R-Modells ist die Idee der Rolle, welche als hauptinformationstragendes Konstrukt fungiert:

Definition 21: Rolle

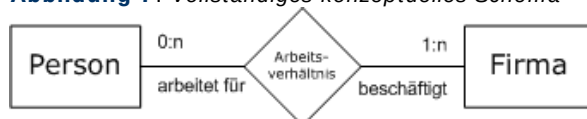
Eine Rolle die durch einen [Entitätstypen](#) innerhalb eines [Assoziationstypen](#) eingenommen wird charakterisiert die konkrete Verwendung von [Entitäten](#) des gegebenen Typs im Kontext der [Assoziationen](#) die zum betrachteten [Assoziationstyp](#) zusammengefaßt werden.

Anmerkung: Für den relationalen Datenbankentwurf ist es notwendig, daß jeder im konzeptuellen Schema modellierte [Entitätstyp](#) entweder über einen [Repräsentationstyp](#) verfügt oder über eine *Namenskonvention*, d.h. eine binäre [Assoziationstyp](#) deren [Kardinalitätsintervalle](#) ausschließlich auf 1:1 festgelegt sind, die den Entitätstyp direkt oder transitiv mit einem mit Repräsentation versehenen Entitätstypen verbindet.

Gleichzeitig wird durch die Rolle der Brückenschlag zwischen natürlicher Sprache und formaler graphischer Darstellung im E³R-Modell ermöglicht.
So lassen sich die Sätze

- Jede Person arbeitet optional für mehrere Firmen.
- Jede Firma beschäftigt ein oder mehrere Personen.

in das nachfolgende konzeptuelle Schema überführen:

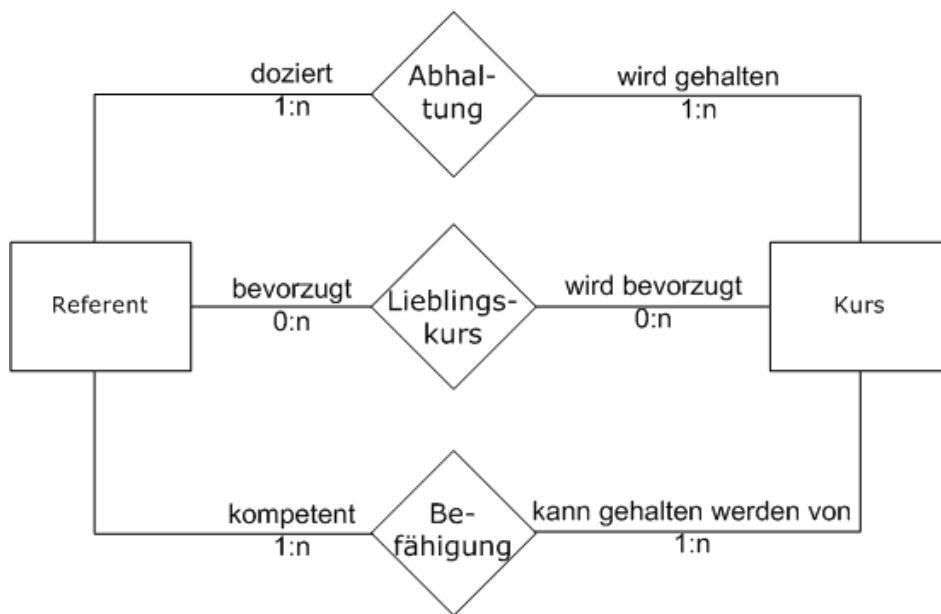
Abbildung 7: Vollständiges konzeptuelles Schema

(click on image to enlarge!)

Zusätzlich zeigt das konzeptuelle Schema der [Abbildung 8](#) die Mächtigkeit des Rollenkonzepts zur Darstellung verschiedener Informationszusammenhänge.

So enthält das abgebildete konzeptuelle Schema die drei verschiedenen Assoziationstypen Abhaltung, Lieblingskurs und Befähigung welche ausschließlich Rollen enthalten die durch die beiden dargestellten Entitätstypen Referent und Kurs gespielt werden.

Abbildung 8: Verschiedene Rollen

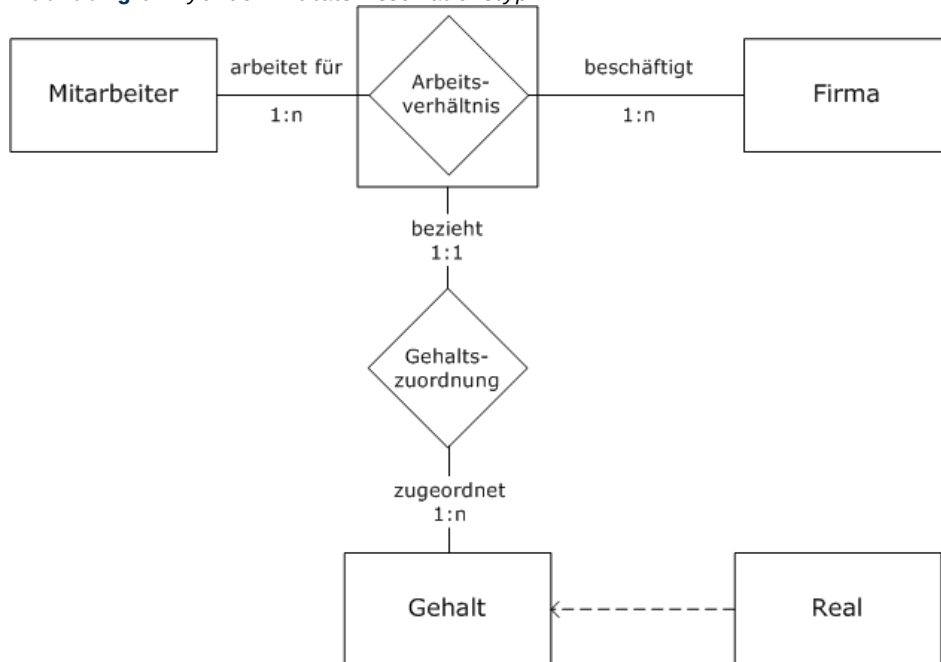


(click on image to enlarge!)

Definition 22: Hybrider Entitäts-Assoziationstyp

Ein hybrider Entitäts-Assoziationstyp vereinigt das Sprachelement des [Entitätstyps](#) und des [Assoziationstyps](#) in sich und bewahrt die Semantik beider Konstrukte.

Abbildung 9: Hybrider Entitäts-Assoziationstyp



(click on image to enlarge!)

Abbildung [Abbildung 10](#) stellt die Informationsstruktur einer Adresse dar.

Dabei zeigt das konzeptuelle Schema die Verwendung der hybriden Entitäts-Assoziationstypen. So können jedem Straßennamen beliebig viele Hausnummern zugeordnet werden und umgekehrt. Jeweils zwei dieser Angaben zusammen bilden die Straße.

Jedem Ortsnamen kann über mehrere Postleitzahlen verfügen, ebenso kann dieselbe Postleitzahl mehreren gleich benannten Orten zugeordnet werden (Beispiel: Ortsteile).

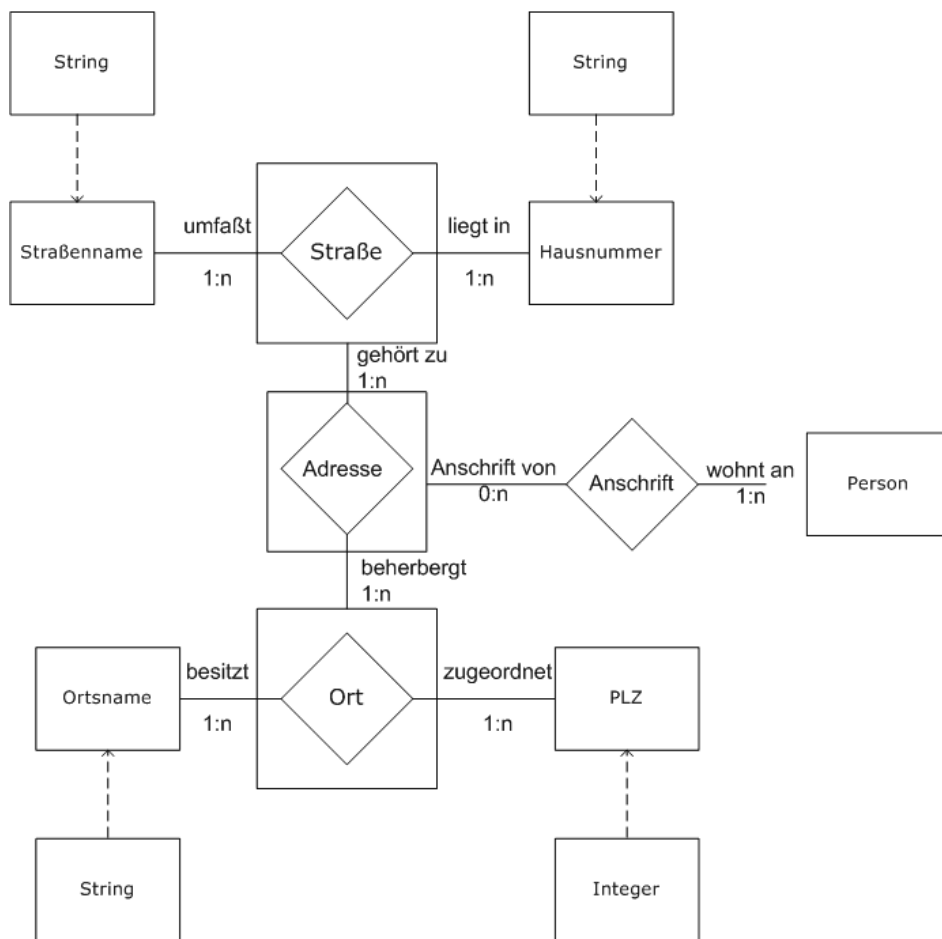
Postleitzahl und Ortsname zusammen bilden den Ort.

Aus der Kombination von Straße und Ort wird eine Adresse gebildet. Dabei kann jede Straße (=Kombination aus Straßennamen und Hausnummer) mehreren Orten (=Kombination aus Ortsname und Postleitzahl) und umgekehrt zugeordnet sein.

Das Beispiel unterstreicht die alleinige Bildbarkeit hybrider Entitäts-Assoziationstypen beim Vorliegen von Kardinalitätsintervallen, die alle über ein Maximum größer 1 verfügen.

Vgl. hierzu Aussagen der [Anmerkung zur Bildung von Kardinalitätsintervallen](#)

Abbildung 10: Informationsstruktur Adresse



(click on image to enlarge!)

Weiterführende Konzepte

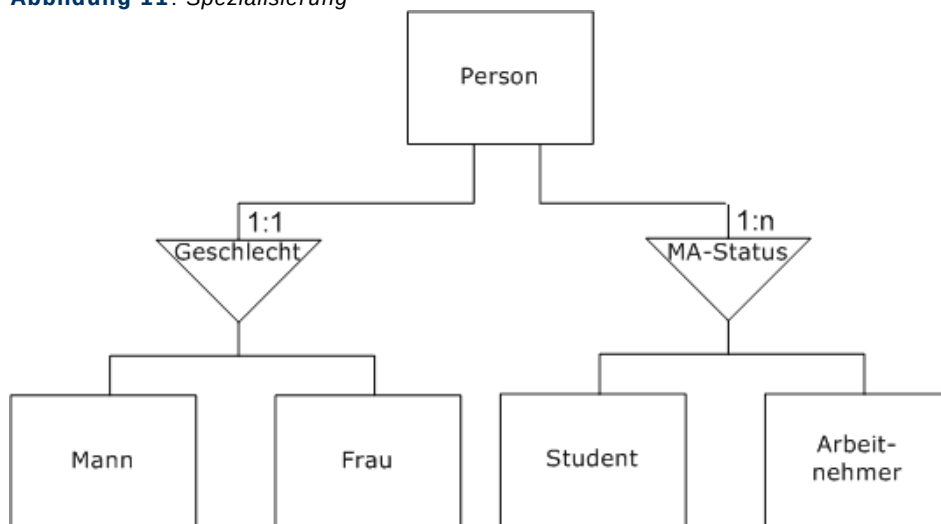
Definition 23: Spezialisierungsassoziationstyp

Ein Spezialisierungsassoziationstyp ist eine duplatfreie ungeordnete Sammlung von logisch als zusammengehörig betrachteten [Assoziationen](#). Zu den in der Menge enthaltenen Assoziationen tragen der zu spezialisierende [Entitätstyp](#) (der sog. *Super- oder Obertyp*) und der spezialisierende (entsprechend als sog. *Sub- oder Untertyp* bezeichnet) Rollen bei. Die Rolle des Supertyps ist hierbei auf *wird spezialisiert zu* fixiert, als Kardinalitätsintervalle sind ausschließlich 0:1, 0:n, 1:1 und 1:n zulässig.

Die Rolle des Subtyps ist auf *ist Spezialisierung von* mit dem Kardinalitätsintervall 1:n fixiert. Jeder Spezialisierungsassoziationstyp wird durch ein Distinktionsmerkmal charakterisiert, das expliziert hinsichtlich welchen Merkmals die Spezialisierung gebildet wird.

Die Verknüpfung durch einen Spezialisierungsassoziationstyp bewirkt, daß alle Assoziations- und Repräsentationstypen, die für den Supertyp definiert sind auch automatisch für alle Subtypen definiert werden.

Abbildung 11: Spezialisierung



(click on image to enlarge!)

Die [Abbildung 11](#) zeigt die Spezialisierung des Entitätstypen Person hinsichtlich des Distinktionsmerkmals Geschlecht und MA-Status. Dabei gilt: Jede Person kann nur mindestens und höchstens einmal (1:1) hinsichtlich ihres Geschlechts zu Mann oder Frau spezialisiert werden. Jede Person kann ein oder mehrfach (1:n) hinsichtlich ihres MA-Status (Mitarbeiter-Status) zu Student und/oder Arbeitnehmer spezialisiert werden.

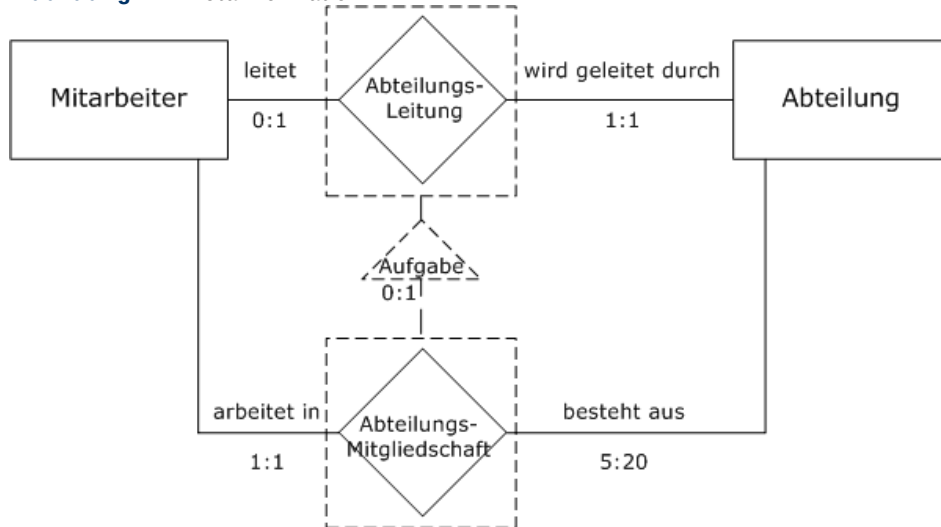
Definition 24: Metainformation

Informationsanteile eines Modells, die nicht direkt in das logische Schema übernommen werden, sondern in konsistenzgarantierende Regeln oder Applikationscode abgebildet werden.

Die [Abbildung 12](#) veranschaulicht die Nutzung des Spezialisierungsassoziationstyps zur Formulierung von Metainformation. Im Beispiel wird gefordert, daß jeder Abteilungsleiter auch gleichzeitig als Mitarbeiter der durch ihn geleiteten Abteilung erfaßt sein muß.

Hinweis: Metainformation muß nicht zwingend semantisch irreduzibel erfaßt werden, wie die --- eigentlich illegale Bildung der beiden hybriden Entitäts-Assoziationstypen Abteilungsleitung und Abteilungsmitgliedschaft zeigt.

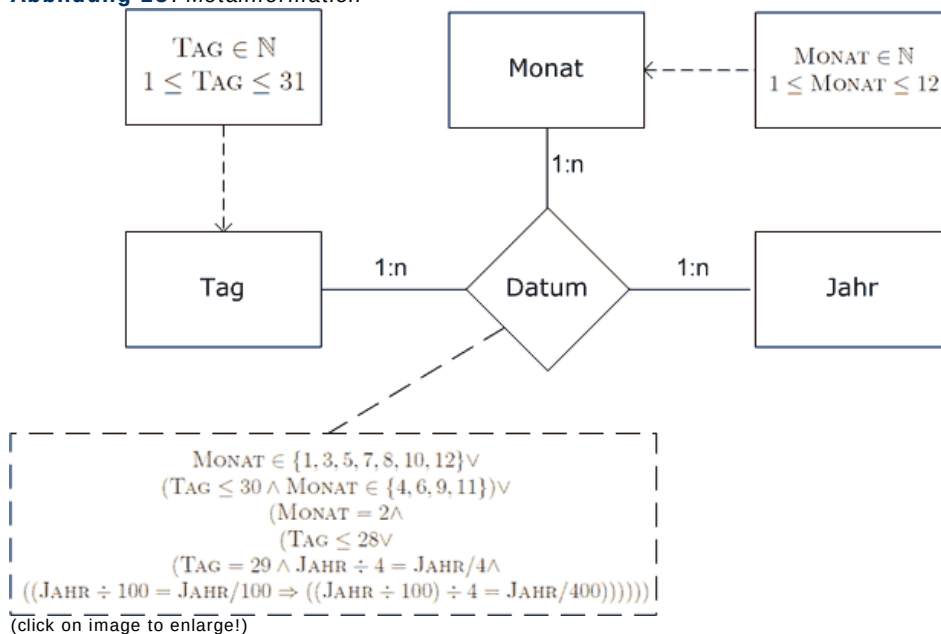
Abbildung 12: Metainformation



(click on image to enlarge!)

Das konzeptuelle Schema der [Abbildung 13](#) zeigt ein Beispiel für konsistenzgarantierende Metainformation, die nicht durch E³R-Syntax ausdrückbar ist und daher in textueller Form annotiert wird.

Abbildung 13: Metainformation



(click on image to enlarge!)

Phasenmodell der Erstellung eines konzeptuellen Schemas mit E³R

E³R ist eine Notation und Methode zur Entwicklung des konzeptuellen Schemas für jede beliebige Art von Kommunikationssituationen.

Phase 0 ist die Vorbereitungsphase, die von der Idee, ein E³-Schema für einen [Realitätsausschnitt](#) zu erstellen, über die Auswahl der Beteiligten bis zu ihrer Ausbildung in den Techniken zur Darstellung von Information und in der Vorgehensweise der Analyse reicht.

Es folgt die Festlegung des Informationsbereichs (**Phase 1**). Hier werden die für den gewünschten Anwendungsbereich relevanten Entitäten und Assoziationen gesammelt und zu Entitäts- und Assoziationstypen zusammengefaßt. Eine große Hilfe dabei sind verbale Beschreibungen der in der Datenbank zu verwaltenden Information, kommentierte Listen mit Daten des betrachteten Informationsbereichs oder ähnliches. Das Ergebnis ist eine erste, grobe Struktur der relevanten Information.

Diese Struktur wird in **Phase 2** immer weiter verfeinert, wobei man für jeden Entitätstyp entweder direkt oder transitiv eine Repräsentation definiert.

Dann werden alle relevanten Eigenschaften, die eine Entität eines Typs haben kann, in Form von semantisch irreduzibel formulierten Assoziationstypen beschrieben. Dabei treten erfahrungsgemäß neue, zuvor nicht berücksichtigte Entitätstypen auf.

Deshalb wird die Phase 2 solange inkrementell iteriert, bis keine neuen Entitätstypen mehr identifiziert werden zu denen noch Repräsentation zu definieren oder durch Assoziationstypen anzubinden sind.

Bis zu diesem Punkt standen strukturelle, formale Gesichtspunkte im Vordergrund. In **Phase 3** treten diese zurück; nun stehen semantische Gesetzmäßigkeiten im Vordergrund, soweit diese nicht bereits in den Phasen 1 und 2 erkannt und behandelt worden sind.

Ziel der Phase 3 ist es, die bis dato erstellte Informationsbeschreibung geeignet zu ergänzen um auch alle nicht durch die E³R-Notation darstellbaren Konsistenzregeln zu erfassen. Zusätzlich kann die E³R-Notation zur Formulierung von Metainformation auf einer höheren Modellebene angewendet werden.

Hinweis: Es kann beim Erstellen des konzeptuellen Schemas durchaus vorkommen, daß sich das Ergebnis der vorausgegangenen Phase als unvollständig herausstellt. In diesem Fall ist es unbedingt notwendig, in diese Phase zurückzukehren und dann mit dem korrigierten Ergebnis dieser Phase weiterzuarbeiten. Dies ist kein Wegwerfen der bisher geleisteten Arbeit, denn meist genügen einige wenige Streichungen und Ergänzungen.

Bleibt diese Regel unberücksichtigt, so nimmt begibt man sich der Möglichkeit wichtige Eigenschaften der Information im konzeptuellen Schema eindeutig festzuhalten. Dabei spricht das Verhältnis zwischen der gewonnenen Exaktheit und dem zusätzlichen Aufwand sehr zugunsten der exakten und sauberen Lösung.

Resultat der korrekten Anwendung des Phasenmodelles ist ein *vollständiges konzeptuelles Schema* als Voraussetzung der Umsetzbarkeit in beliebige logische Strukturen.

Definition 25: *Vollständiges konzeptuelles Schema*

Ein vollständiges konzeptuelles Schema ist ein E³R-Schema in dem alle Entitäts- und Assoziationstypen, sowie alle Rollen benannt sind. Darüberhinaus ist jede Rolle mit einem korrekten Kardinalitätsintervall versehen, sowie jedem Entitätstypen direkt oder transitiv ein Repräsentationstyp zugeordnet.

Sofern Metainformation existiert, ist diese auch in adäquater Weise dargestellt.

Fallstudie: Fächerdatenbank

Der Fachbereich möchte die Belegung der Fächer in einer Datenbank abspeichern; hierfür gelten folgende semantische Regeln:

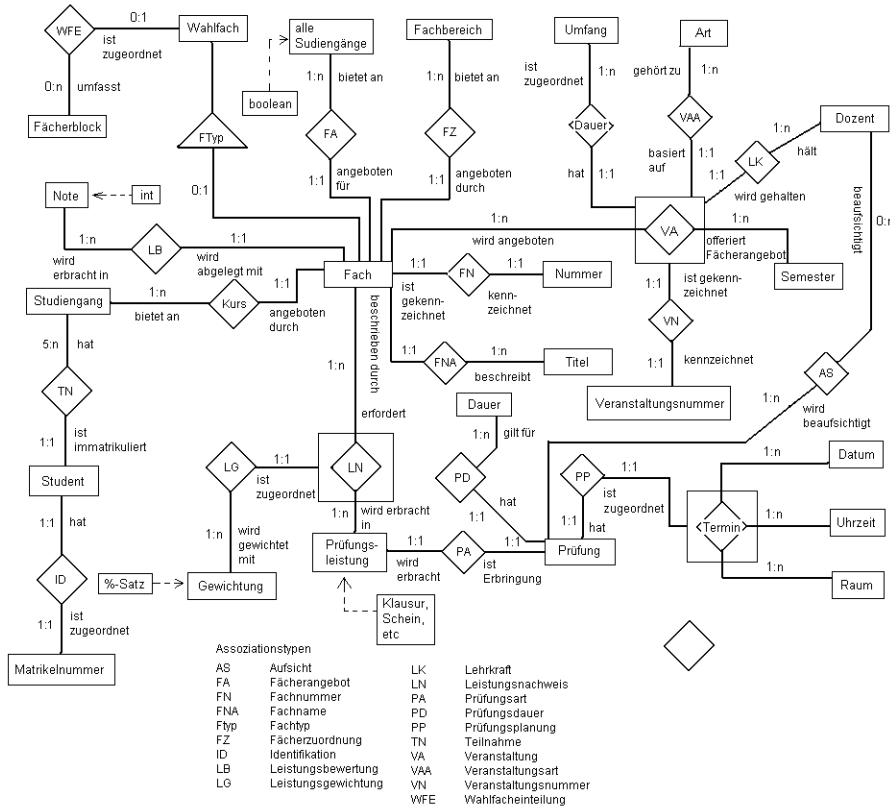
Die vorgesehenen Fächer haben eine feste Nummer, die sich niemals ändert sowie eine einen längeren Titel. Zusätzlich sind sie von einem Fachbereich entweder für einen speziellen Studiengang (z.B. *WIB*) oder allen Studiengängen angeboten, dann gilt die Zuordnung *FH*. Gleichzeitig kann jedes Wahlfach einem Fächerblock (z.B. *Consulting* oder *Informatik*) zugeordnet sein.

Die in einem bestimmten Semester angebotenen Fächer erhalten eine Veranstaltungsnummer, die nur für dieses Semester gilt. Dazu wird der jeweilige Dozent angegeben und die Art der Lehrveranstaltung (*Vorlesung*, *Seminar*, *Praktikum* etc.) sowie ihr Umfang in Semesterwochenstunden.

Die Notenbildung in jedem Fach kann durch eine oder mehrere Prüfungsleistungen erfolgen (z.B. *Leistungsnachweis*, *Schein*, *Prüfung*, etc.), die in unterschiedlichen Prozentsätzen gewichtet werden. Jede Prüfung findet zu einem festgelegten Datum in einem Raum zu einer Uhrzeit statt und wird durch mindestens einen Dozenten beaufsichtigt. Zusätzlich soll die Dauer der Prüfungsleistung vermerkt werden.

Ein Student ist durch eine eindeutige Matrikelnummer gekennzeichnet; weiter werden von ihm noch Name, Geburtsdatum und Studiengang gespeichert.

Abbildung 14: Konzeptuelles Schema der Fallstudie



(click on image to enlarge!)

2.2 Ableitung logischer Relationenstrukturen

Erweiterung der Grundbegriffe des Relationenmodells

Definition 26: Superschlüssel

Ein Superschlüssel SK ist eine nicht-leere Teilmenge von Attributen einer [Relation](#) für die gilt, daß zwei verschiedene Tupel t_1 und t_2 dieser Relation keine gleiche Wertbelegung aufweisen.

Der Superschlüssel definiert damit eine Eindeutigkeitseinschränkung, nach der zwei Tupel allein über die Betrachtung der im Superschlüssel zusammengefaßten Attribute unterscheidbar sind.

Gegeben sei die Relation *Mitarbeiter*:

Vorname	Nachname	Geburtsdatum	Persausweisnummer
Xaver	Oberm•ller	1970-03-04	134975459
Rosi	Hinterhuber	1973-06-02	781367519
Rosi	Oberm•ller	1963-11-03	783148384
Hans	Hinterhuber	1970-03-04	977554422

Mögliche Superschlüssel dieser Relation sind:

- (Vorname, Nachname, Personalausweisnummer)
- (Nachname, Geburtsdatum, Personalausweisnummer)
- (Geburtsdatum, Personalausweisnummer)
- ...

Beispiel 14: Beispiele für Superschlüssel**Definition 27:** *Schlüssel*

Ein Schlüssel K ist ein [Superschlüssel](#), der sofern man ein Attribut aus ihm entfernt nicht mehr eindeutigkeitseinschränkend wirkt.

Der einzige Schlüssel der Relation [Mitarbeiter](#) ist Personalausweisnummer.

Beispiel 15: Beispiele für Schlüssel

Anmerkungen:

- Die Menge aller Attribute einer Relation ist immer Superschlüssel.
- Jeder Schlüssel ist auch ein Superschlüssel.
Der Umkehrschluß gilt nicht, da Schlüssel eine schärfere Forderung darstellt.
- Jeder Schlüssel ist zwingend eine minimal identifizierende Attributkombination.

Häufig tritt es in der Praxis auf, daß sich in einer Relation mehr als ein Schlüssel finden läßt. Jeder dieser möglichen gleichwertigen Schlüssel wird daher als *Schlüsselkandidat* bezeichnet.

Gegeben sei die Relation *Lagerverwaltung*:

Lagerplatz	Produktnummer	Produktname	Menge
7952	7946	Wusch Superfein	3
7412	9854	Blitzbank Extra	5
7894	6542	Maiengrün natur	7
9461	8954	Gelber Gigant	5

Die Relation enthält folgende Schlüsselkandidaten.

- Lagerplatz
- Produktnummer
- Produktname

Beispiel 16: Beispiele für Superschlüssel**Definition 28:** *Primärschlüssel*

Ein Primärschlüssel P ist ein [Schlüssel](#), der als identifizierendes Merkmal ausgewählt wurde.

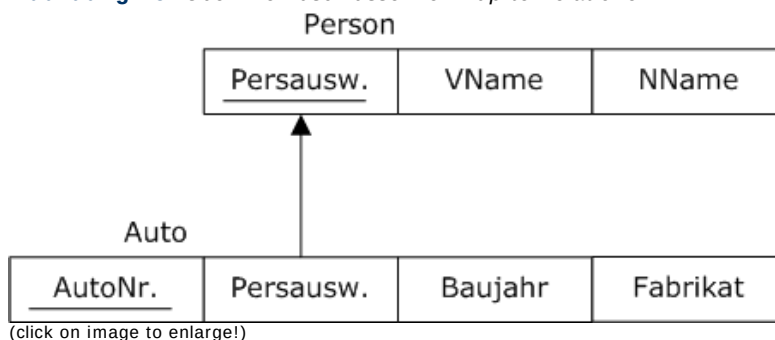
In der [graphischen Darstellung der Demo-DB](#) sind die Primärschlüssel durch Unterstreichung der beitragenden Attribute hervorgehoben.

Zur Wahrung der Konsistenz innerhalb einer relationalen Datenbank wird üblicherweise u.a. das Mittel der *referentiellen Integrität* eingesetzt, um gleicher Wertinhalte in Attributen (derselben oder verschiedener Relationen) aufeinander abzustimmen.

Definition 29: *Referentielle Integrität*

Attributwerte einer durch referentielle Integrität verknüpften Relation müssen auch in der verknüpften Relation existieren.

Abbildung 15: Über Fremdschlüssel verknüpfte Relationen



Das Attribut Persausw. der Relation Auto verweist auf den Primärschlüssel gleichen Namens der Relation Person.

Als Konsequenz dürfen für Persausw. in Auto nur Werte definiert werden, die sich bereits im Attribut Persausw. von Person finden.

Beispiel 17: Beispiel für referentielle Integrität

Die Prüfung von Primärschlüsselwerten erfordert u.U. eine Reihe zusätzlicher Datenbankzugriffe. Zu ihrer Beschleunigung können zusätzliche Speicherbereiche, sog. *Indexe* angelegt werden.

Definition 30: Index

Ein Index ist ein zusätzlicher Speicherbereich der in der Datenbank verwaltet wird um den lesenden Zugriff auf einzelne Tupel zu beschleunigen.

In der Konsequenz der Beschleunigung der lesenden Zugriffe durch zusätzlichen Speicherplatz verringert sich die Geschwindigkeit der schreibenden Zugriffe (Erzeugung, Aktualisierung und Löschung) etwas.

Die Tabelle tab besteht aus zwei Attributen UUID1 und UUID2, wobei ersteres duplikatfrei indexiert wird.

Aktion	Dauer ohne Index [sec]	Dauer mit Index [sec]
Einfügen von 10.000.000 Tupeln INSERT INTO tab VALUES (...)	1812	2025
Auswahl aller Tupel SELECT COUNT(*) FROM tab WHERE UUID1<> "X" (UUID1 enthält niemals den Wert X daher werden alle Tupel selektiert)	2100	1800
Auswahl genau eines Tupels SELECT UUID2 FROM tab WHERE UUID1="..."	0,422	0,028
Aktualisierung genau eines Tupels UPDATE tab SET UUID2="Z" WHERE UUID1="..."	0,415	0,033
Aktualisierung keines Tupels, jedoch vollständige Durchsuchung eines Attributs. UPDATE tab SET UUID2="Z" WHERE UUID1<>"X"	0,395	0,014
Löschung eines Tupels DELETE FROM tab WHERE UUID1="..."	0,431	0,043
Löschung keines Tupels, jedoch vollständige Durchsuchung eines Attributs. DELETE FROM tab WHERE UUID1="X"	0,037	0,008

Beispiel 18: Geschwindigkeitsverhalten mit/ohne Index

Die Erstellung des Index nimmt, bei in der Tabelle gehaltenen 10.000.000 Tupeln 363,063 Sekunden in Anspruch.

Definition 31: NULL-Wert

Fehlende Attributwerte in einer Relation werden durch den gesonderten Datenbankeintrag NULL dargestellt.

Für die Wertbelegung NULL stellt das DBMS sicher, daß sie nicht mit der Ziffer 0 oder dem leeren String kollidiert.

Der Algorithmus

Zentrale Zielsetzung der Erstellung des konzeptuellen Schemas ist die Möglichkeit von ihm ausgehend unterschiedliche logische Modelle, die später in die physische Implementierungssicht abgebildet werden, ableiten zu können.

Dieser Abschnitt stellt einen Algorithmus vor, der es erlaubt aus dem mit E³R formulierten

konzeptuellen Schema logische Strukturen gemäß dem Relationenmodell abzuleiten.

Dabei operiert der Algorithmus ausschließlich auf der graphischen Repräsentation des konzeptuellen Schemas und kann daher auch von entsprechend ausgebildeten Fachexperten manuell durchgeführt werden.

Der durch den Algorithmus abgeleitete logische Datenbankentwurf orientiert sich an festgelegten Gütekriterien um einen redundanzfreien und somit anomalienfreien Entwurf zu gewährleisten.

Schritt 1

Markiere alle Verbindungslinien, an denen das Kardinalitätsintervall 1:1 steht.

Anmerkung: Eine ausschließliche 1:1-Markierung stellt einen logisch korrekten DB-Entwurf sicher.

Das zusätzliche Markieren aller 0:1-Verbindungen führt zu Performanceverbesserungen. Bei relationalen DBMS, die fehlende Werte (NULL) zulassen führt das Markieren von 0:1-Verbindungen zu einem optimalen relationalen DB-Entwurf. Bei Implementierungen die auch optionale Schlüsselkandidaten zulassen führt das vorgesetzte Markieren über 0:1 Verbindungen hinweg zu effizienten DB-Strukturen.

Schritt 2

1. Bilde die Zusammenhangskomponenten (bestehend aus Entitäts-, Assoziations- und Spezialisierungsassoziationstypen) bezüglich der markierten Verbindungslinien.
2. Übrigbleibende Zusammenhangskomponenten bestehen entweder aus genau einem Entitäts- oder Assoziationstyp:
 - Falls das Mitgliedschaftsintervall an der Verbindung zwischen einem solchen Entitätstyp und irgendeinem Assoziationstyp den minimalen Wert 0 hat, ist aus diesem Entitätstyp eine eigenständige Zusammenhangskomponente zu bilden, sofern der Entitätstyp kein Repräsentationstyp oder die entsprechenden Entitäten nicht schon in einem anderen Assoziationstyp definiert sind (Kardinalitätsintervall 1:z; $z \geq 1$).
 - Die zu Untertypen führenden Kanten werden behandelt wie gewöhnliche Assoziationstypen.
 - Repräsentationstypen werden nicht berücksichtigt.
 - Übrigbleibende Assoziationstypen bilden jeweils eine eigene Zusammenhangskomponente.

Schritt 3

Treten innerhalb einer Zusammenhangskomponente Zyklen auf, so sind diese wie folgt zu behandeln:

Anmerkung: Ein Zyklus in einer Zusammenhangskomponente ist eine Folge $\{ET_1, AT_{1,2}, ET_2, AT_{2,3}, \dots, ET_n, AT_{n,1}\}$ mit den Eigenschaften:

- In allen Assoziationstypen $_{i,k}$ ($1 \leq i, k \leq n$) wird die eine Rolle von ET_i und die andere Rolle von ET_k gespielt.
- Anmerkung:* Zu Untertypen führende Kanten werden behandelt wie gewöhnliche Beziehungen zu Assoziationstypen.
- Alle Verbindungslinien einer Folge sind markiert.

Zur Ableitung von Relationen müssen Zyklen aufgelöst werden:

- Falls innerhalb eines Zyklus 0:1-Markierungen vorhanden sind, werden diese gelöscht;
- falls nur 1:1-Markierungen vorhanden sind, wird eine beliebig festzusetzende Verbindungslinie gelöscht.

Schritt 4

Aus jeder Zusammenhangskomponente wird eine Relation nach folgenden Regeln:

1. Namensgebend für eine Relation ist genau einer der innen liegenden Entitätstypen.

- Enthält eine Zusammenhangskomponente nur einen Assoziationstyp, so bekommt die abgeleitete Relation dessen Namen.
2. Die Relation enthält je ein Attribut für jeden direkt mit einer Repräsentation versehenen Entitätstypen im Inneren der Zusammenhangskomponente. Ein Entitätstyp wird zusammen mit seinen sämtlichen Untertypen als ein Entitätstyp betrachtet, sofern die Untertypen über keine eigene Repräsentation verfügen.
 3. Zusätzlich enthält die Relation für jeden Assoziationstyp im Inneren einer Zusammenhangskomponente noch je ein Attribut für jede innenliegende Rolle, die ein Entitätstyp spielt, der außerhalb der Zusammenhangskomponente liegt.
Für einen im Inneren der Zusammenhangskomponente liegenden Assoziationstyp sind alle Entitätstypen, zu denen eine nicht markierte Verbindungslinie führt, außerhalb.
 4. Zusammenhangskomponenten, die ausschließlich aus genau einem Assoziationstyp bestehen werden in eine eigenständige Relation überführt, die für jede zum Assoziationstyp beitragende Rolle ein Attribut enthält.
Dieses Attribut wird mit der Repräsentation des rollenspielenden Entitätstypen typisiert.
 5. Jedes aus einem Entitätstyp im Inneren einer Zusammenhangskomponente abgeleitete Attribut ist Schlüsselkandidat.
Außerdem sind alle diejenigen Attribute Schlüsselkandidaten, deren entsprechende Kardinalitätsintervalle das Maximum 1 besitzen.
 6. In den restlichen Fällen sind alle Attribute zusammen Schlüsselkandidat.
 7. Existiert in einer Relation mehr als genau ein Schlüsselkandidat, so ist einer unter diesen als Primärschlüssel auszuzeichnen.
 8. Befindet sich ein Untertyp nicht in derselben Zusammenhangskomponente wie seine Obertypen, so erbt er jeweils die Repräsentation.
 9. Relationen, deren Attribute sich aus jeweils gleichen Rollen ableiten, werden durch eine einzige Relation dargestellt.
Treten in einer Zusammenhangskomponente gleiche Rollen eines Entitätstyps mehrfach auf, so werden sie in einer entsprechenden Relational als genau ein Attribut übernommen.
 10. Manuelles Eingreifen:
bei 0:1-Markierung ist u.U. eine Entscheidung, orientiert an der modellierten Semantik, zu treffen:
Folgende Konstellationen können das Rückgängigmachen von Markierungen innerhalb einer Zusammenhangskomponente notwendig werden lassen:
mehrere Schlüsselkandidaten und:
 - alle Schlüsselkandidaten sind optional
 - nicht alle verpflichtend und die Notwendigkeit vorhanden, einen bestimmten als Primärschlüssel festzulegen.

Schritt 5

1. Leiten sich aus einem Entitätstyp mehrere sich entsprechende Attribute ab, so sind die folgenden Abhängigkeiten (Fremdschlüsselbeziehungen) zu berücksichtigen:
 - Ist eine Attributkombination Schlüsselkandidat, so sind zu den entsprechenden Attributkombinationen, die als Primärschlüssel ausgewählt wurden, Fremdschlüsselbeziehungen vorzusehen.
Hinweis: Fremdschlüsselbeziehungen bedeuten zusätzliche Zugriffe und sollten daher in der Datenbank entsprechend durch Indexstrukturen unterstützt werden.
 - Beim Auftreten identischer Schlüsselkandidaten sind ebenfalls Fremdschlüsselbeziehungen vorzusehen.
2. Metainformationen, die nicht die DB-Strukturebene betreffen, sondern Ausprägungen einschränken, werden den entsprechenden DB-Strukturelementen zugeordnet (z.B. Domäneneinschränkungen bei Attributen).

Schritt 6

Ist ein Entitätstyp Spezialisierung (d.h. Untertyp) eines anderen, so wird in die Relation die aus der Zusammenhangskomponente gebildet wurde, welche den Untertypen beinhaltet die Primärschlüsselattribute derjenigen Relation übernommen, die den Obertypen beinhaltet.

Anmerkung: Schlüsselkandidaten dieser Relation werden identisch zu den anderen Relationen ermittelt.

Schritt 7

1. Systemunabhängig

- Alle Attribute bekommen als Datentyp den Entitätstyp, durch den sie repräsentiert werden.
- Bei 1:1-Markierung wird für alle Attribute der Relation ein NOT NULL vergeben.
- Ein Primärschlüssel muß stets mit NOT NULL vereinbart werden.
- Bei markierten 0:1-Beziehungen: Alle aus über 0:1-Beziehungen angebotenen Entitätstypen entstehenden Attribute werden auf NULL gesetzt.
- Schlüsselkandidaten und Zugriffspfade werden als Indexe angelegt.
- Soweit für Metainformation formale Umsetzungsmöglichkeiten existieren, werden die entsprechenden konsistenzgarantierenden Einschränkungen formuliert.

2. Systemabhängig

Die Syntax für die physische Realisierung der Relationen (Tabellen) und der Indexstrukturen sowie der Datentypen und Einschränkungen (soweit unterstützt) müssen dem jeweiligen DBMS angepaßt werden.

Evtl. durch das DBMS automatisch angelegte Indexstrukturen müssen nicht mehr explizit formuliert werden.

2.3 Algebraischer Entwurf mit der Normalformtheorie

Neben dem graphischen Entwurf logischer DB-Strukturen genießt der algebraische Entwurf auf Basis der sog. *Normalformtheorie* in Theorie und Praxis große Bedeutung. Historisch gesehen stellt die Betrachtung von relationalen Strukturen mit Hilfe mathematischer Methoden die älteste Disziplin dar und findet sich heute in allen bedeutenden Lehrbüchern. Dieser Abschnitt führt in die sechs verschiedenen Normalformen hinsichtlich ihrer Definition sowie ihrer Implikationen auf die Struktur des logischen Modells ein und zieht Parallelen zur Vorgehensweise des eingeführten Algorithmus zur Umsetzung konzeptueller Strukturen des E³R-Modells.

Ebenso wie der Algorithmus zur Transformation eines E³R-Schemas führt auch die Normalisierung zu einem anomalienfreien relationalen Datenbankentwurf. Voraussetzung der Anomaliefreiheit ist die konsequente Ermittlung und Eliminierung von Redundanz, d.h. keine Information darf in der Datenbank mehrfach vorhanden sein.

Insgesamt verfolgt der Normalisierungsprozeß folgende Ziele:

- Redundanzvermeidung als Basis der Anomaliefreiheit
- Vermeidung unnötiger Abhängigkeiten, die Performanceeinbußen bei Einfüge-, Löscho- und Änderungsoperationen nach sich ziehen
- Senkung der Anzahl beteiligter Tabellen bei der Modifikation der Datenbank
- Erhöhung des Dokumentationsgrades des entstehenden Datenmodells (Ziel: Verständlichkeit)

Die Anomaliefreiheit ist die zentrale Basisforderung und Zielsetzung des Normalisierungsprozesses. Im Detail werden drei Ausprägungen unterschiedlicher Anomalien unterschieden:

- **Einfügeanomalie:** Durch das Hinzufügen eines korrekten neuen Tupels werden konsistent vorliegende Daten in einen inkonsistenten Zustand überführt.
- **Löchanomalie:** Durch die Entfernung eines Tupels entsteht ein inkonsistenter Datenbestand.
- **Aktualisierungsanomalie:** Durch die Änderung eines vorhandenen Tupels entsteht ein inkonsistenter Datenbestand.

Alle Arten von Anomalien gehen auf das Vorhandensein von Redundanz, mithin einem Verstoß gegen die Grundregel jede im konzeptuellen Schema modellierte Information an nur genau einer Stelle abzuspeichern, zurück.

Ausgangssituation des Normalisierungsvorganges ist die *Urrelation* die alle Attribute in genau einer Relation zusammenfaßt.

Erste Normalform

Definition 32: Erste Normalform (1NF)

Eine Relation ist dann in erster Normalform, wenn ihre Domänen (=Wertausprägungen der Attribute) nur einfache (atomare) Werte besitzen.

Atomarer Wert bedeutet hierbei, daß kein Attributinhalt strukturiert sein darf, d.h. durch mögliche Zerlegungsoperationen in kleine eigenständige Informationseinheiten zerlegt werden kann.

Beispiel einer Relation die **nicht in 1NF** ist:

FNAME	LNAME	ADDRESS	BDATE
John	Smith	731 Fondren, Houston, TX	1965-01-09
Franklin	Wong	638 Voss, Houston, TX	1955-12-08
Joyce			1972-07-31
Polly Esther	Wallace	291 Berry, Bellaire, TX	1941-06-20, 1952-09-04

Beispiel 19: Relation, die nicht in 1NF ist

Ziel der Überführung in 1NF ist es, Relationen zu erhalten, die in gängigen RDBMS abspeicherbar sind. Diese bieten zwar heute technische Mechanismen (wie Array- und Referenztypen) an, die Strukturen ähnlich den dargestellten verwaltbar werden lassen. Voraussetzung ihrer konzeptionellen Beherrschung ist jedoch die vorherige Normalisierung.

Im Beispiel befinden sich die Zeilen von Franklin und Joyce Wong nicht in 1NF, da sie nicht für jedes Attribut einen Wert besitzen, sondern sich eine Wertausprägung (Wong und 638 Voss, Houston, TX) teilen.

Ebenso befindet sich der Eintrag von Polly Esther Wallace nicht in 1NF, da hier für das Geburtsdatum unerlaubterweise zwei Einträge auftreten.

Die beiden Vornamen sind im Rahmen der Semantik des Attributes FNAME zugelassen und daher im Normalisierungsprozeß nicht zu beanstanden.

Die Relation aus [Beispiel 19](#) in erster Normalform:

FNAME	LNAME	ADDRESS	BDATE
John	Smith	731 Fondren, Houston, TX	1965-01-09
Franklin	Wong	638 Voss, Houston, TX	1955-12-08
Joyce	Wong	638 Voss, Houston, TX	1972-07-31
Polly Esther	Wallace	291 Berry, Bellaire, TX	1941-06-20
Polly Esther	Wallace	291 Berry, Bellaire, TX	1952-09-04

Beispiel 20: Relation, die in 1NF ist

Zur Umformung der Relation in eine Relation in erster Normalform wurden die „gemeinsamen“ Attribute aufgelöst, so das nunmehr jedes Attribut genau einem Tabelleintrag (Tupel) zugeordnet ist.

Zusätzlich wurde für jedes Attribut die atomare Belegung sichergestellt.

Die Einführung der ersten Normalform verhindert damit die Bildung *geschachtelter Relationen*, die entstünden, jedes Attribut eine Menge anderer Attribute, mithin wiederum eine vollständige Relation, enthalten könnte.

Anmerkung: Diese Forderung wird durch die in SQL:1999 definierten ARRAY-Typen aufgeweicht und für postrelationale und objektorientierte Datenbanken vollständig aufgegeben, weshalb diese Strukturen auch als *Non-First-Normal-Form* (kurz: NFNF, NF2 oder NF²) bezeichnet werden.

Test auf Einhaltung der ersten Normalform:

Die Relation sollte keine nicht-atomaren Attribute oder verschachtelte Relationen enthalten.

Der algorithmische Ableitungsprozeß aus dem konzeptuellen Schema stellt durch die Organisation der [Repräsentationstypen](#) sicher, daß Attribute ausschließlich durch atomare Werte repräsentiert werden.

Zweite Normalform

Grundlegende Voraussetzung zum Verständnis der zweiten Normalform ist das Konzept der *vollen funktionalen Abhängigkeit*.

Definition 33: *Volle funktionale Abhängigkeit*

Ein Attribut y einer Relation ist vollfunktional abhängig von einem Attribut x wenn gilt, daß jede Ausprägung von x genau eine Ausprägung von y bestimmt und y nicht abhängig von Teilattributen von x ist.

Im Zeichen: $x \rightarrow y$.

Die vollfunktionale Abhängigkeit wird häufig als *FD (functional dependency)* abgekürzt.

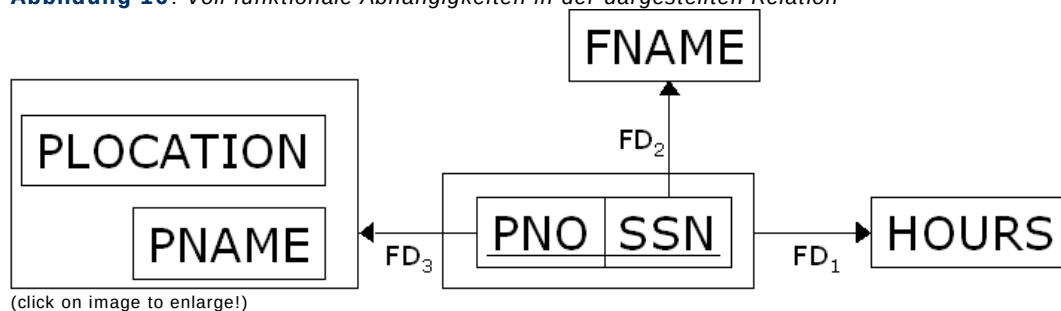
Bemerkung: Man beachte, daß der Begriff *Attribut* in [Definition 33](#) eine nichtleere Menge von Attributen bezeichnet.

Beispiel aus der Demodatenbank:

SSN	PNO	HOURS	FNAME	DNAME	PNAME	PLOCATION
123456789	1	32.5	John	Research	ProductX	Bellaire
123456789	2	7.5	John	Research	ProductY	Sugarland
333445555	2	10.0	Franklin	Research	ProductY	Sugarland
333445555	3	10.0	Franklin	Research	ProductZ	Houston
333445555	10	10.0	Franklin	Research	Computerization	Stafford
333445555	20	10.0	Franklin	Research	Reorganization	Houston
453453453	1	20.0	Joyce	Research	ProductX	Bellaire
453453453	2	20.0	Joyce	Research	ProductY	Sugarland
666884444	3	40.0	Ramesh	Research	ProductZ	Houston
888665555	20	NULL	James	Headquarters	Reorganization	Houston
987654321	20	15.0	Jennifer	Administration	Reorganization	Houston
987654321	30	20.0	Jennifer	Administration	Newbenefits	Stafford
987987987	10	35.0	Ahmad	Administration	Computerization	Stafford
987987987	30	5.0	Ahmad	Administration	Newbenefits	Stafford
999887777	10	10.0	Alicia	Administration	Computerization	Stafford
999887777	30	30.0	Alicia	Administration	Newbenefits	Stafford

In der Relation existieren folgende voll funktionale Abhängigkeiten:

Abbildung 16: *Voll funktionale Abhängigkeiten in der dargestellten Relation*



Es ist offensichtlich, daß zwar HOURS vom vollständigen Primärschlüssel (gebildet aus PNO

gemeinsam mit SSN) abhängen (FD_1), aber der Name (FNAME) und die Kombination aus PLOCATION und PNAME nur von SSN bzw. PNO und damit Teilen des Primärschlüssels abhängen (FD_2 bzw. FD_3).

Inhaltlich manifestieren sich diese Probleme im Zwang verschiedene Daten (etwa SSN und FNAME) wiederholt (redundant) abspeichern zu müssen. Ändert sich eine dieser Angaben, so muß potentiell eine große Anzahl Tupel in der Datenbank aktualisiert werden. Werden hierbei nicht alle Datensätze aktualisiert so entsteht ein inkonsistenter Datenbestand. Zusätzlich ist es nicht möglich bestimmte Informationszusammenhänge abzubilden. Hierunter fällt beispielsweise der Wunsch Projekte (etwa: PLOCATION und PNAME) zur verwalten, denen noch keinen Mitarbeiter zugeordnet ist.

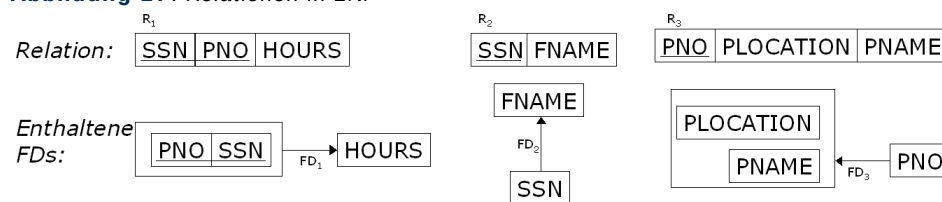
Um eine Relation in 2NF zu überführen muß sie so zerlegt werden, daß jede entstehende neue Relation ausschließlich genau eine voll funktionale Abhängigkeit beinhaltet.

Definition 34: Zweite Normalform (2NF)

Eine Relation ist in 2NF genau dann, wenn sie in 1NF ist und jedes Nichtschlüsselattribut voll funktional abhängig von einem Schlüsselkandidaten ist.

Entstehende Relationen:

Abbildung 17: Relationen in 2NF



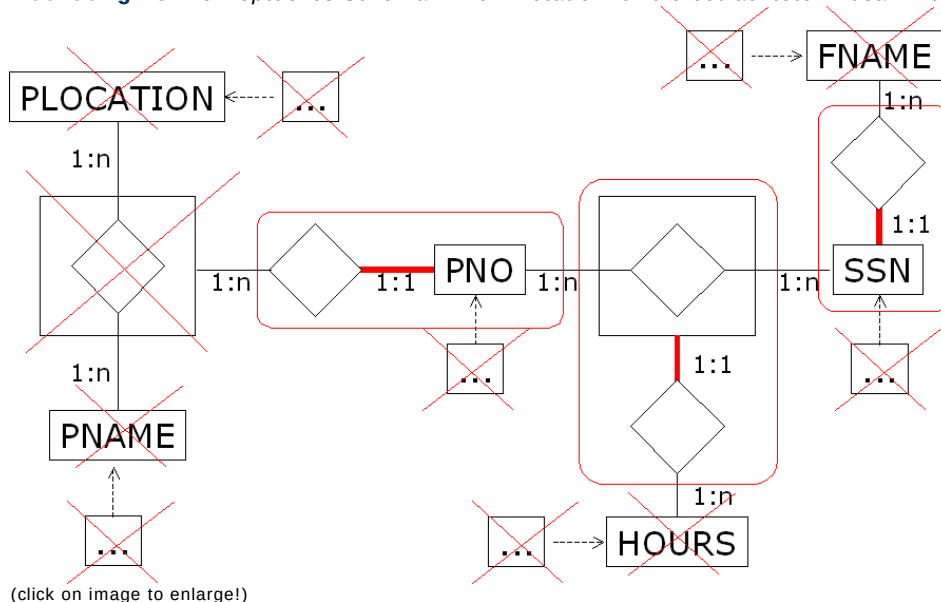
(click on image to enlarge!)

Test auf Einhaltung der zweiten Normalform:

In Relationen deren Primärschlüssel mehrere Attribute enthalten, sollte kein Nichtschlüsselattribut voll funktional von einem Teil des Primärschlüssels abhängen.

Der Ableitungsprozeß aus dem konzeptuellen Schema in E³R-Notation gewährleistet automatisch die Erzeugung von Relationen in 2NF:

Abbildung 18: Konzeptuelles Schema in E³R-Notation für die betrachteten Zusammenhänge



Dritte Normalform (3NF)

Die dritte Normalform erweitert die für die Zweite getroffenen Aussagen dahingehend, daß zusätzlich zur voll funktionalen Abhängigkeit die transitive Abhängigkeit eingeführt und betrachtet wird.

Definition 35: Transitive Abhängigkeit

In einer Relation R ist ein Attribut z transitiv von einem Attribut x abhängig dann und nur dann, wenn z voll funktional von y und y voll funktional von x abhängig ist.

Im Zeichen: x->->z

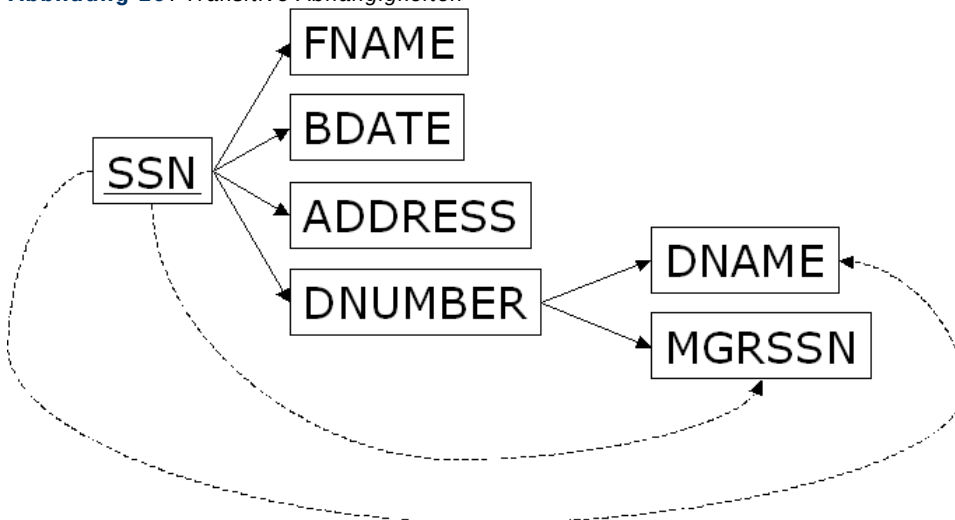
Bemerkung: Man beachte, daß der Begriff *Attribut* in [Definition 35](#) eine nichtleere Menge von Attributen bezeichnet.

Im Beispiel der Demodatenbank:

FNAME	SSN	BDATE	ADDRESS	DNUMBER
DNAME	MGRSSN			
James	888665555	1937-11-10	450 Stone, Houston, TX	1
Headquarters	888665555			
Jennifer	987654321	1941-06-20	291 Berry, Bellaire, TX	4
Administration	987654321			
Ahmad	987987987	1969-03-29	980 Dallas, Houston, TX	4
Administration	987654321			
Alicia	999887777	1968-07-19	3321 Castle, Spring, TX	4
Administration	987654321			
John	123456789	1965-01-09	731 Fondren, Houston, TX	5
Research	333445555			
Franklin	333445555	1955-12-08	638 Voss, Houston, TX	5
Research	333445555			
Joyce	453453453	1972-07-31	5631 Rice, Houston, TX	5
Research	333445555			
Ramesh	666884444	1962-09-15	975 Fire Oak, Humble, TX	5
Research	333445555			

In der Relation existieren folgende direkten und transitive Abhängigkeiten (die direkten funktionalen Abhängigkeiten sind durch gerichtete Kanten mit durchgezogener Linienführung, die Transitiven durch unterbrochene Linienführung dargestellt):

Abbildung 19: *Transitive Abhängigkeiten*



(click on image to enlarge!)

In dieser Organisationsform tritt eine [Einfügeanomalie](#) auf, wenn ein Mitarbeiter neu eingefügt wird und dabei „falsche“ (d.h. inkonsistente) Werte für die Abteilung angelegt werden. Zusätzlich fällt das Einfügen neuer Abteilungen, zu denen (noch) kein Mitarbeiter abgespeichert wird, schwer, da SSN zwingend anzugeben ist (NULL-Wert ist wegen der Definition als Primärschlüssel nicht zugelassen!).

Daneben existiert eine [Löschanomalie](#) dahingehend, daß wenn der letzte Mitarbeiter einer Abteilung aus der Datenbank entfernt wird auch alle Informationen über diese Abteilung verloren gehen.

Werden Abteilungsdaten verändert, so kann es zu einer [Modifikationsanomalie](#) kommen, da nicht in jedem Falle eindeutig klärbar ist ob nur die Abteilungsdaten dieses Mitarbeiters verändert werden sollen (beispielsweise im Falle eines Abteilungswechsels) oder die Daten aller abgespeicherten Abteilungen (beispielsweise im Falle der Umbenennung einer Abteilung).

Die dritte Normalform setzt sich zum Ziel die Ursachen dieser Anomalien zu beseitigen:

Definition 36: Dritte Normalform (3NF)

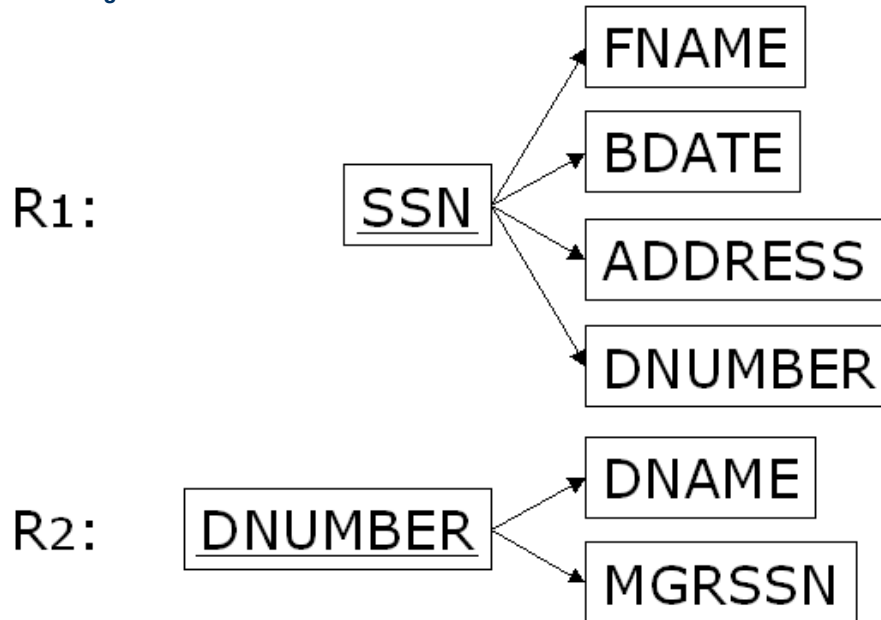
Eine Relation ist dann und nur dann in 3NF, wenn sie in 2NF ist und jedes Nicht-Schlüsselattribut nicht-transitiv abhängig ist von einem Schlüsselkandidaten; sie ist auch in 3NF, wenn sich eine transitive Abhängigkeit ausschließlich über den Schlüsselkandidaten herleiten läßt.

Gemäß dieser Definition ist die Beispielrelation zu zerlegen in:

$R_1(\underline{SSN}, FNAME, BDATE, ADDRESS, DNUMBER)$ und

$R_2(\underline{DNUMBER}, DNAME, MGRSSN)$.

Abbildung 20: Relation in 3NF



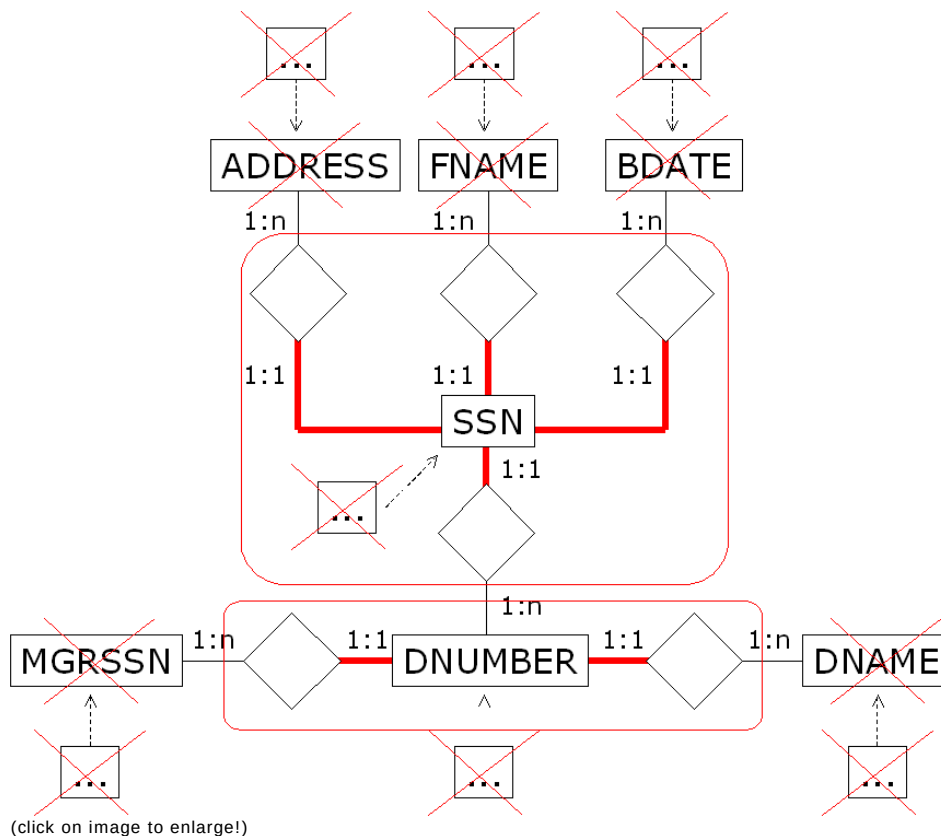
(click on image to enlarge!)

Test auf Einhaltung der dritten Normalform:

Eine Relation sollte kein Nicht-Schlüsselattribut enthalten, das voll funktional von einem anderen Nicht-Schlüsselattribut (oder von einer Menge von Nichtschlüsselattributen) abhängig ist. Das heißt, es sollte keine transitive Abhängigkeit eines Nichtschlüsselattributs vom Primärschlüssel bestehen.

Der Ableitungsprozeß aus dem konzeptuellen Schema in E³R-Notation gewährleistet automatisch die Erzeugung von Relationen in 3NF:

Abbildung 21: Konzeptuelles Schema in E³R-Notation für die betrachteten Zusammenhänge



Das Rissanen Theorem klärt die sich prinzipiell ergebene Fragestellung warum die in [Abbildung 1](#) dargestellte Zerlegung gewählt wurde, da sich prinzipiell unter Ausnutzung der transitiven Abhängigkeiten auch eine (Alternativ-)Zerlegung in $R_1(\text{SSN}, \text{FNAME}, \text{BDATE}, \text{ADDRESS})$ und $R_2(\text{SSN}, \text{DNAME}, \text{MGRSSN})$ angeboten hätte.

Nach dem Theorem von Heath und Rissanen ist jedoch die in [Abbildung 20](#) gewählte Zerlegung die einzig korrekt mögliche, da sie die tatsächlich vorhandenen funktionalen Abhängigkeiten erhält, was bei obiger Alternative nicht der Fall wäre.

Angewendet auf unser Beispiel bedeutet dies, daß für obige (nach Heath/Rissanen fehlerhafte) Zerlegung beispielsweise der Anwendungsfall nach Anlage einer Abteilung (DNUMER gemeinsam mit ihrem Namen (DNAME) und der SSN ihres Abteilungsleiters (MGRSSN) nicht möglich wäre, da DNAME und MGRSSN nur verwaltet werden können, wenn gleichzeitig die als Primärschlüssel zwingend anzugebende SSN eines Mitarbeiters dieser Abteilung existiert. Mithin wäre die Speicherung einer Abteilung die nur über ihren Leiter aber (noch) nicht über Mitarbeiter verfügt nicht möglich.

Die gewählte Zerlegung vermeidet jedoch diese Einschränkung und liefert, ebenso wie der Algorithmus aus dem konzeptuellen Schema, das korrekte Resultat.

Boyce/Codd-Normalform (BCNF)

Ursprünglich wurde die *Boyce/Codd Normalform* (BCNF) als Vereinfachung der [dritten Normalform](#) vorgeschlagen. Jedoch faßt sie diese schärfer und führt so zu einem neuen Typ von Normalform, der nach ihren Schöpfern Boyce und Codd benannt wurde.

Inhaltlich räumt sie mögliche Anomalien aus, die in Relationen, welche sich bereits in 3NF befinden, noch auftreten können.

Definition 37: Boyce/Codd-Normalform

In einer Relation ist dann und nur dann in BCNF, wenn sie in [3NF](#) ist und gleichzeitig jede Determinante Schlüsselkandidat ist.

Hierbei definiert die BCNF den Begriff der *Determinante* als den Ausgangspunkt einer funktionalen Abhängigkeit.

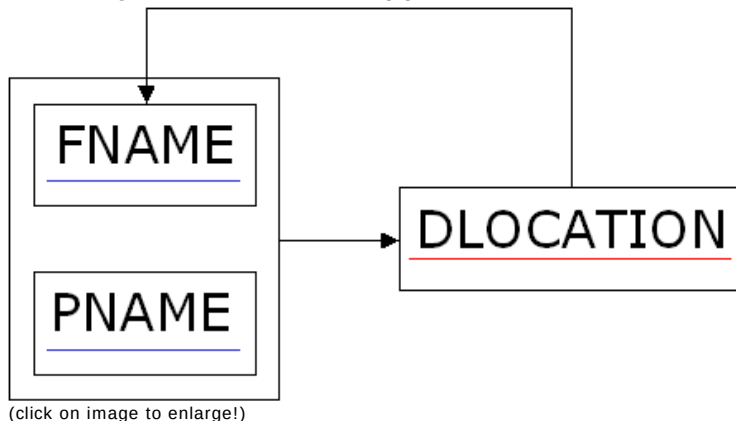
Im Beispiel der Demodatenbank:

FNAME	PNAME	DLOCATION
...	...	...

Franklin	ProductY	Sugarland
Franklin	ProductZ	Houston
Jennifer	ProductY	Bellaire
Jennifer	Newbenefits	Stafford

In der Relation existieren folgende funktionale Abhängigkeiten:

Abbildung 22: Funktionale Abhängigkeiten



Die Schlüsselkandidaten sind durch farbliche Unterstreichung hervorgehoben. Hierbei bestimmt FNAME gemeinsam mit PNAME eindeutig einen Wert für DLOCATION (blau) und DLOCATION (rot) bestimmt eindeutig einen Wert für FNAME.

Da sich die transitive Abhängigkeit $PNAME \rightarrow DLOCATION \rightarrow FNAME$ ausschließlich über Schlüsselkandidaten herleitet ist die Relation in 3NF.

Dennoch ist sie nicht anomaliefrei, da sie beispielsweise die Ablage von Abteilungsstandorten (DLOCATION) und den Namen der Abteilungsleiter (FNAME) verbietet, wenn kein Projektname (PNAME) zusätzlich abgespeichert wird.

Ebenso ist die Ablage von Projekten und den zuständigen Abteilungen erst dann möglich, wenn zusätzlich der Name des Abteilungsleiters bekannt ist.

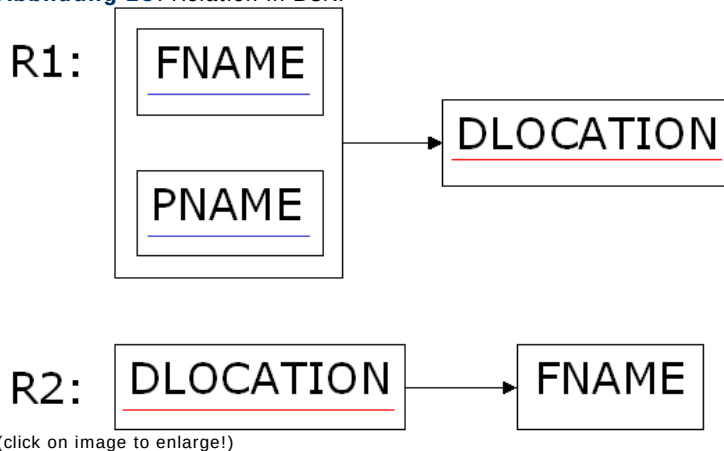
Dieses Manko wird durch die Forderung der BCNF behoben. Sie erzwingt die Zerlegung der abgebildeten Ausgangsrelation in zwei Eigenständige. Hierbei wird jede Determinante der Ausgangsrelation zum Schlüsselkandidaten in den neu gebildeten Relationen.

Gemäß [Definition 37](#) ist die Beispielrelation zu zerlegen in:

$R_1(FNAME, PNAME, DLOCATION)$

$R_2(DLOCATION, FNAME)$.

Abbildung 23: Relation in BCNF

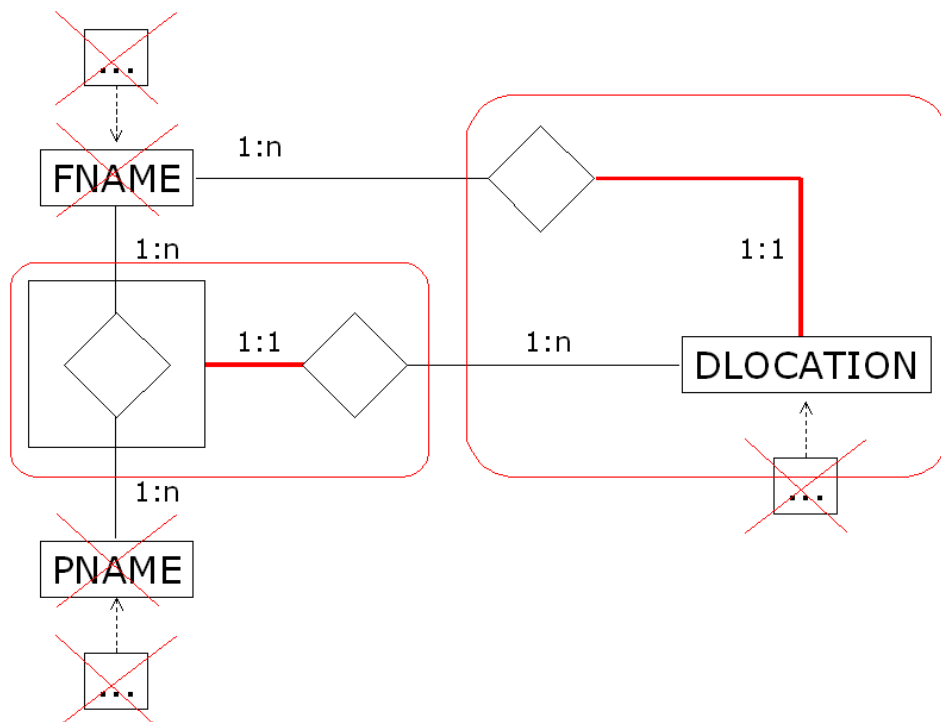


Test auf Einhaltung der Boyce/Codd-Normalform:

Eine Relation sollte lediglich direkt vom Schlüssel vom Schlüssel abhängende Attribute enthalten.

Der Ableitungsprozeß aus dem konzeptuellen Schema in E³R-Notation gewährleistet automatisch die Erzeugung von Relationen in BCNF:

Abbildung 24: Konzeptuelles Schema in E³R-Notation für die betrachteten Zusammenhänge



Vierte Normalform (4NF)

Die bisher betrachteten Normalformen nutzen alle das Vorhandensein funktionaler Abhängigkeiten aus. Jedoch existieren neben diesem Beziehungstyp auch noch andere, im vorhergehenden nicht berücksichtigte, Abhängigkeit.

Definition 38: Mehrwertige Abhängigkeit

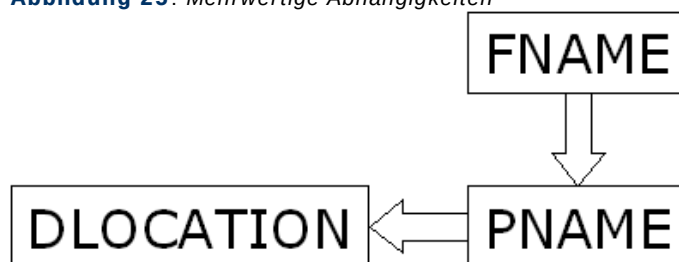
Eine mehrwertige Abhängigkeit (*multivalue dependency*, MVD) ist eine Abhängigkeit, die einem Attribut eine Menge verschiedener Werte zuordnet.

Das Vorhandensein mehrwertiger Abhängigkeiten in einer Relation kann zu Aktualisierungsanomalien führen, wie die Betrachtung des bekannten Beispiels aus der Demodatenbank:

FNAME	PNAME	DLOCATION
Franklin	ProductY	Sugarland
Franklin	ProductZ	Houston
Jennifer	ProductY	Bellaire
Jennifer	Newbenefits	Stafford

In der Relation existieren folgende mehrwertige Abhängigkeiten:

Abbildung 25: Mehrwertige Abhängigkeiten



Die Existenz der dargestellten mehrwertigen Abhängigkeiten in der Datenbank führt dazu, daß dieselbe Information mehrfach abgespeichert werden muß. So enthält die Beispieldatenbank mehrfach denselben FNAME sowie wiederholt denselben Produktnamen (PNAME).

Als Konsequenz dieser Wiederholung müssen bei jedem Aktualisierungsvorgang, der die Zuordnung des Produktes zum FNAME betrifft alle FNAME-Einträge geändert werden, jedoch bei der Zuständigkeitsänderung eines Produktverantwortlichen nur der betroffene FNAME.

Ziel der vierten Normalform ist die Elimination von Redundanzen, die aus mehrwertigen Abhängigkeiten herrühren. Allerdings können mehrwertige Abhängigkeiten nicht vollständig entfernt werden, daher ist für die Normalisierung gemäß 4NF die Eliminierung aller *nicht trivialen* mehrwertigen Abhängigkeiten als Zielsetzung definiert.

Eine *triviale mehrwertige Abhängigkeit* ist hierbei festgelegt als:

Definition 39: *Triviale mehrwertige Abhängigkeit*

Eine triviale mehrwertige Abhängigkeit ist eine mehrwertige Abhängigkeit zwischen Attributmengen X und Y der Relation R für die gilt: Y ist eine Teilmenge von X oder die Vereinigung von X und Y bildet R.

Definition 40: *Vierte Normalform*

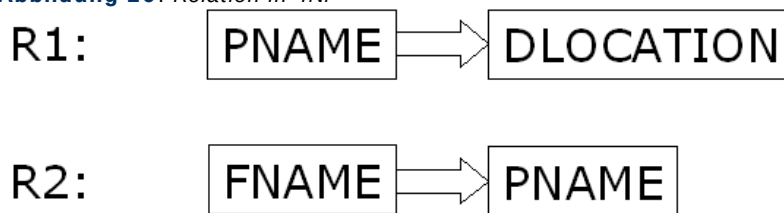
Eine Relation ist dann in vierter Normalform, wenn sie nur noch triviale mehrwertige Abhängigkeiten enthält.

Gemäß dieser Definition ist die Beispielrelation zu zerlegen in:

$R_1(\underline{DLOCATION}, PNAME)$ und

$R_2(\underline{FNAME}, PNAME)$.

Abbildung 26: *Relation in 4NF*



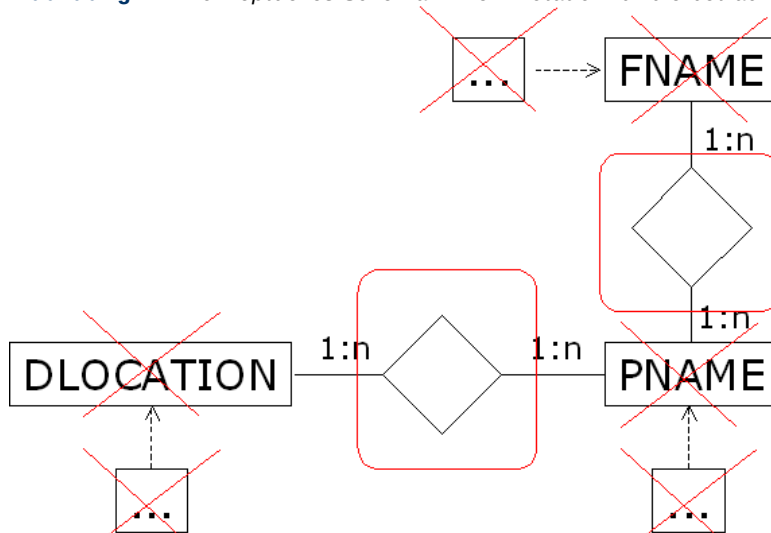
(click on image to enlarge!)

Test auf Einhaltung der vierten Normalform:

Eine Relation sollte keine nicht trivialen mehrwertigen Abhängigkeiten enthalten.

Der Ableitungsprozeß aus dem konzeptuellen Schema in E³R-Notation gewährleistet automatisch die Erzeugung von Relationen in 4NF:

Abbildung 27: *Konzeptuelles Schema in E³R-Notation für die betrachteten Zusammenhänge*



(click on image to enlarge!)

Fünfte Normalform (5NF)

Die fünfte Normalform greift anders als alle vorhergehenden nicht auf die Zusammenhänge der Typebene zurück, sondern benötigt zu ihrer Untersuchung die Betrachtung von tatsächlichen Datenbankinhalten.

Ausgehend von diesen definiert sie die fünfte Normalform als:

Definition 41: *Fünfte Normalform*

Eine Relation ist dann in fünfter Normalform (5NF), bei ihrer Zerlegung durch Projektionen und deren anschließender Kombination durch Verbundoperationen keine Tupel gebildet werden, die nicht Bestandteil der Ausgangsrelation waren.

Aufgrund ihrer Abstützung auf die Projektions- und Verbundoperation (engl. *join*) wird die 5NF auch als *Project Join Normalform* (PJNF) bezeichnet.

Im Beispiel der Demodatenbank:

Relation Ur:

FNAME	DNUMBER	DLOCATION
John	5	Bellaire
John	5	Houston
James	1	Houston

Durch Projektion ergeben sich die drei möglichen Relationen:

SELECT Ur.FNAME, Ur.DNUMBER INTO R11 FROM Ur

R₁₁:

FNAME	DNUMBER
John	5
John	5
James	1

SELECT Ur.FNAME, Ur.DLOCATION INTO R12 FROM Ur

R₁₂:

FNAME	DLOCATION
John	Bellaire
John	Houston
James	Houston

SELECT Ur.DNUMBER, Ur.DLOCATION INTO R13 FROM Ur

R₁₃:

DNUMBER	DLOCATION
5	Bellaire
5	Houston
1	Houston

Durch Verbundoperationen entstehen die Relationen:

SELECT R11.FNAME, R11.DNUMBER, R12.DLOCATION INTO R21 FROM R11 INNER JOIN R12 ON R11.FNAME=R12.FNAME

R₂₁:

FNAME	DNUMBER	DLOCATION
John	5	Bellaire
John	5	Houston
James	1	Houston

SELECT R11.FNAME, R11.DNUMBER, R13.DLOCATION INTO R22 FROM R11 INNER JOIN R13 ON R11.DNUMBER = R13.DNUMBER

R₂₂:

FNAME	DNUMBER	DLOCATION
John	5	Bellaire
John	5	Houston
James	1	Houston


```
SELECT R12.FNAME, R12.DLOCATION, R13.DNUMBER INTO R23 FROM R12 INNER JOIN R13 ON
R12.DLOCATION=R13.DLOCATION
```

R₂₃:

FNAME	DNUMBER	DLOCATION
John	5	Bellaire
John	5	Houston
James	1	Houston
James	5	Houston
John	1	Houston

Trotz der ausschließlich durch Rekombination der zuvor aus der Urrelation gebildeten Projektionen (d.h. ohne Hinzunahme neuer) Daten „entstehen“ in Relation R₂₃ (rot hervorgehobene) Tupel (sog. *spurious tuple*), die in dieser Form nicht in der Ausgangsrelation präsent waren.

Erst das Zusammenfügen aller aus Verbundoperationen erzeugten Relationen durch einen erneuten Verbund eliminiert diesen *unechten Tupel*:

```
SELECT R21.FNAME, R22.DNUMBER, R23.DLOCATION INTO RESULT FROM (R21 INNER JOIN R22
ON (R21.FNAME = R22.FNAME) AND (R21.DNUMBER = R22.DNUMBER) AND (R21.DLOCATION =
R22.DLOCATION)) INNER JOIN R23 ON (R22.FNAME = R23.FNAME) AND (R22.DNUMBER = R23.
DNUMBER) AND (R22.DLOCATION = R23.DLOCATION) AND (R21.FNAME = R23.FNAME) AND (R21.
DNUMBER = R23.DNUMBER) AND (R21.DLOCATION = R23.DLOCATION)
```

FNAME	DNUMBER	DLOCATION
John	5	Bellaire
John	5	Houston
James	1	Houston

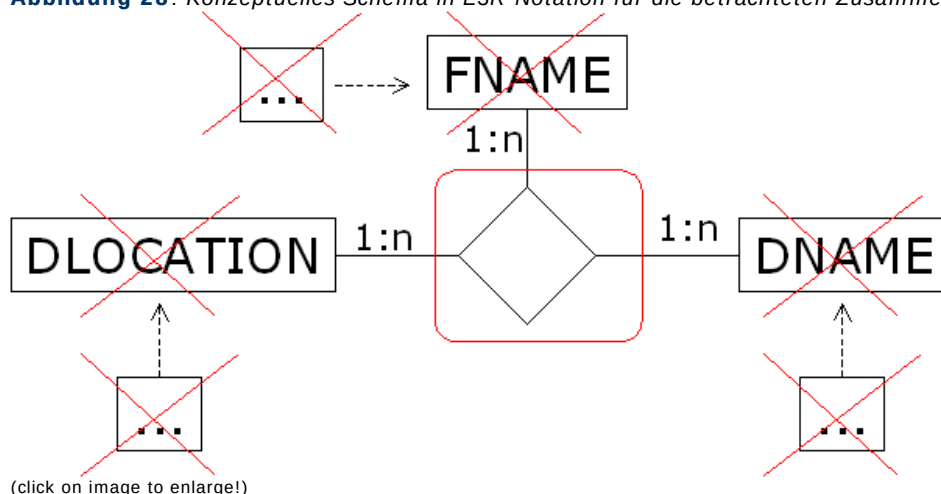
Das Auftreten unechter Tupel ist ein Indiz für eine Relation in der Verbundabhängigkeiten existieren. Eine solche Relation darf nicht weiter zerlegt werden, da durch die Entfernung von Attributen Information verloren ginge.

Test auf Einhaltung der fünften Normalform:

Wenn durch Projektions- und Verbundoperationen keine unechten Tupel entstehen, kann die Relation weiter zerlegt werden, andernfalls ist sie in 5NF.

Der Ableitungsprozeß aus dem konzeptuellen Schema in E³R-Notation gewährleistet automatisch die Erzeugung von Relationen in 5NF:

Abbildung 28: Konzeptuelles Schema in E³R-Notation für die betrachteten Zusammenhänge



Die 5NF ist die höchstmögliche über den Normalisierungsprozeß erreichbare Normalform. Dies bedeutet jedoch nicht, daß jede Relation bis in 5NF gebracht werden kann. Der Normalisierungsprozeß endet generell mit der höchstmöglichen Normalform, dies muß jedoch nicht immer 5NF sein, sondern orientiert sich an den modellierten Informationszusammenhängen.

Weitere Normalisierungsaspekte

Über die im vorhergehenden betrachteten formalisierten Normalformen hinaus existieren noch weitere Abhängigkeiten und Normalformen-ähnliche Güteaussagen für Datenbanken.

Inklusionsabhängigkeit:

Die Inklusionsabhängigkeit (engl. *inclusion dependence*, ID) beschreibt die Beziehungen zwischen Super- und Subtypen, insbesondere die *Attributentsprechung*, d.h. die Tatsache, daß der Subtype mindestens alle Attribute des Supertypen aufweist, sowie die *Kompatibilitätsrelation* worunter die Austauschbarkeit von Ausprägungen der beiden Typen verstanden wird für Anwendungsbereiche die lediglich auf die gemeinsam vorhandenen Attribute zugreifen.

aus der Inklusionsabhängigkeit folgen u.a. die drei Inferenzregeln:

IDIR₁: (Reflexivität) Die Inklusionsabhängigkeit ist reflexiv.

IDIR₂: (Attributentsprechung) Verfügen zwei Typen über dieselben Attribute, dann entsprechen sie sich.

IDIR₃: (Transitivität) Ist *B* Untertyp von *A* und *C* Untertyp von *B*, dann ist auch *C* Untertyp von *A*.
Trotz dieser formalisierbaren Aussagen und der breiten Verwendung von Modellierungskonstrukten zur Darstellung von Spezialisierungsbeziehungen wurden auf Basis der Inklusionsabhängigkeit bisher noch keine Normalformen vorgeschlagen.

Template-Abhängigkeiten:

Die Idee der Template-Abhängigkeit fußt auf der Definition einer Reihe von *Hypothesetupeln* und daraus abgeleiteten *Konklusionstupeln*, welche gültige Ausprägungen der Datenbank abstrahiert beispielhaft aufzeigen.

Template-Abhängigkeiten gestatten die einfache Formulierung von Intrarelationsabhängigkeiten die sich auf konkrete Wertausprägungen einzelner Attribute beziehen.

Das Beispiel zeigt die Abhängigkeit, daß kein Angestellter mit mehr Einkommen ausgestattet sein darf als sein Vorgesetzter:

EMPLOYEE(FNAME, SSN ... , SALARY, SUPERSSN)	
	a b ... c d
Hypothese	e f ... g h

Konklusion	c < g

Domain-Key-Normalform (DKNF):

Grundannahme der Domain-Key-Normalform (DKNF) ist es, daß wenn für jedes Attribut einer Relation eine Domäne (d.h. die Menge der zugelassenen Wertbelegungen) angegeben wird, alle Änderungsanomalien verschwinden.

Gleichzeitig fordert die DKNF die eindeutige Identifikation jedes Attributs einer Relation durch einen Schlüssel.

In der Praxis kann jedoch die Angabe einer allgemein formulierten eindeutigen Domäne mitunter zu Schwierigkeiten führen, weshalb die Prüfung auf Einhaltung dieser Normalform mit erheblichen technischen Umsetzungsschwierigkeiten verbunden ist.

3 Arbeiten mit einer Datenbank

3.1 Codd'sche Regeln und Eigenschaften relationaler Systeme

Trotz des in dieser Hinsicht sehr eindeutigen grundlegenden Papiers von E. F. Codd über die Relationenstruktur (*A Relational Model of Data for Large Shared Data Banks*) existierte lange Zeit keine Übereinkunft darüber welche Eigenschaften ein relationales DBMS mindestens aufweisen muß um dieser Systemklasse zugerechnet werden zu können.

Daher definiert Codd 1986 in einem zweiteiligen Artikel für die Zeitschrift *Computer World* 12 strenge Regeln die ein RDBMS aus seiner Sicht zwingend erfüllen muß um als solches eingestuft werden zu können.

Diese hierin erhobenen Forderungen sind jedoch so streng, daß sie bis heute kein System vollständig erfüllt.

Die Regeln sind nachfolgend mit ihren englischsprachigen Originalbezeichnungen wiedergegeben, da sich für sie bisher keine eindeutige und allgemeinverständliche deutsche Übersetzung etablieren konnte.

Regel 1: **The Information Rule:**

Alle Daten, die in einer Datenbank gespeichert werden sind auf dieselbe Art dargestellt, nämlich durch Werte in Tabellen.

Anmerkung: In dieser Definition wurde bewußt der Begriff der [Tabelle](#) gegenüber dem der [Relation](#) bevorzugt.

Regel 2: Guaranteed Access Rule:

Jeder gespeicherte Wert muß über Tabellenname, Spaltenname und Wert des Primärschlüssels zugreifbar sein, wenn der zugreifende Anwender über hinreichende Zugriffsrechte verfügt.

Regel 3: Systematic Treatment of Null Values:

Nullwerte müssen datentypunabhängig zur Darstellung fehlender Werte unterstützt werden. *Systematisch* drückt hierbei aus, daß Nullwerte unabhängig von demjenigen Datentyp für den sie auftreten gleich behandelt werden.

Regel 4: Dynamic On-line Catalog Based on the Relational Model: Forderung nach einem Online-Datenkatalog (*data dictionary*) in Form von Tabellen.

Dieser Katalog beschreibt die in der Datenbank abgelegten Tabellen hinsichtlich ihrer Struktur und zugelassenen Inhaltsbelegungen.

Regel 5: Comprehensive Data Sublanguage Rule:

Für das DBMS muß mindestens eine Sprache existieren durch die sich die verschiedenen Inhaltstypen (Tabelleninhalte, Sichten, Integritätsstrukturen (Schlüsselbeziehungen, Wertebereichseinschränkungen, Aufzählungstypen) sowie Zugriffsrechte) definieren lassen.

Regel 6: View Updating Rule:

Sofern theoretisch möglich, müssen Inhalte von Basistabellen auch über deren Sichten änderbar sein.

Regel 7: High-level Insert, Update, and Delete:

Innerhalb einer Operation können beliebig viele Tupel bearbeitet werden, d.h. die Operationen werden grundsätzlich mengenorientiert ausgeführt. Hierfür ist eine so abstrahierte Sicht dieser Operationen notwendig, daß keinerlei Information über die systeminterne Darstellung der Tupel notwendig ist.

Regel 8: Physical Data Independence:

Änderungen an der internen Ebene dürfen keine Auswirkungen auf die auf den abgespeicherten Daten operierenden Anwendungsprogramme besitzen.

Werden Daten demnach reorganisiert oder beispielsweise durch [Indexe](#) zugriffsbeschleunigt, so darf eine solche Änderung die auf die Datenbank zugreifenden Anwendungsprogramme nicht beeinträchtigen.

Regel 9: Logical Data Independence:

Änderungen des [konzeptuellen Schemas](#) dürfen keine Auswirkung auf die Anwendungsprogramme besitzen, solange diese nicht direkt von der Änderung betroffen sind.

Regel 10: Integrity Independence:

In Verfeinerung der fünften Regel wird gefordert, daß alle Integritätsbedingungen ausschließlich durch die Sprache des DBMS definieren lassen können müssen. Definierte Integritätsbedingungen müssen in Tabellen abgespeichert werden und durch das DBMS zur Laufzeit abgeprüft werden.

Im Mindesten werden folgende Forderungen durch verfügbare Systeme unterstützt:

- Kein Attribut welches Teil eines Primärschlüssels ist darf NULL sein.
- Ein Fremdschlüsselattribut muß als Wert des zugehörigen Primärschlüssels existieren.

Regel 11: Distribution Independence:

Die Anfragesprache muß so ausgelegt sein, daß Zugriffe auf lokal gehaltene Daten identisch denen auf verteilt gespeicherte Daten formuliert werden können.

Hieraus läßt sich auch die Ausdehnung der Forderungen nach logischer und physischer Datenunabhängigkeit für verteilte Datenbanken ableiten.

Regel 12: Nonsubversion Rule:

Definiert ein DBMS neben der High-level Zugriffssprache auch eine Schnittstelle mit niedrigerem Abstraktionsniveau, dann darf durch diese keinesfalls eine Umgehung der definierten Integritätsregeln möglich sein.

Zusätzlich faßt Codd in **Regel 0** nochmals die Anforderungen dahingehend zusammen, daß er postuliert, alle Operationen für Zugriff, Verwaltung und Wartung der Daten ausschließlich mittels relationaler Fähigkeiten abzuwickeln.

Derzeit existiert kein am Markt verfügbares kommerzielles System welches alle zwölf Regeln vollständig umsetzt. Insbesondere sind die Regeln 6, 9, 10, 11 und 12 in der Praxis schwer umzusetzen.

3.2 Implementierung des logischen Modells mit SQL-DDL

Die SQL-DDL dient allgemein der Definition und Verwaltung von Tabellen- und Indexdefinitionen innerhalb einer relationalen Datenbank entlang ihres gesamten Lebenszyklus, d.h. von ihrer Erstellung über alle Wartungsstadien bis hin zur Entfernung.

Nicht normiert durch den SQL-Standard sind die notwendigen Schritte zur Erzeugung einer Datenbank innerhalb eines Datenbankmanagementsystems. Überdies variieren die hierfür abzusetzenden Kommandos von Hersteller zu Hersteller und müssen der spezifischen Dokumentation entnommen werden.

Hinweis: Zwar läßt SQL inzwischen die beliebige Schreibung der Schlüsselworte (groß, klein oder gemischt) zu, zur bessern Hervorhebung und Kompatibilität mit existierender Literatur werden sie jedoch in den nachfolgenden Syntaxübersichten und Beispielen durchgehend in Großschreibung wiedergegeben.

Innerhalb der Syntaxbeschreibungen gelten folgende Konventionen:

- Schlüsselworte, die direkt wie abgedruckt eingegeben werden müssen sind großgeschrieben.
- Optionale Bestandteile, die weggelassen werden können sind in eckigen Klammern („[]“) dargestellt.
Die Klammern selbst sind nicht Bestandteil der Syntax und müssen nicht eingegeben werden.
- Dargestellte runde Klammern („()“) sind Syntaxbestandteil und müssen unverändert eingegeben werden.
- Senkrechte Striche („|“) trennen Alternativen von denen jeweils eine ausgewählt werden kann, nicht jedoch mehrere.
- Kommentare, die auf die Ausführung keinen Einfluß haben werden durch zwei Minuszeichen („-“) eingeleitet und enden mit dem Zeilenende.
- Alle SQL-Befehle werden generell durch ein Semikolon abgeschlossen. Dieses ist aus Übersichtlichkeitsgründen in den Syntaxübersichten weggelassen.

Erzeugen von Tabellen

Die SQL-Anweisung CREATE TABLE dient der Erzeugung neuer Tabellen innerhalb einer bestehenden Datenbank. Sie legt die Struktur und die zugelassenen Typausprägungen, sowie Einschränkungen hinsichtlich der erlaubten Werte fest.

Die vereinfachte Syntax der CREATE TABLE-Anweisung lautet:

```
CREATE [TEMPORARY] TABLE tbl_name [(create_definition,...)]
[table_options] [select_statement]
```

create_definition:

```
col_name type [NOT NULL | NULL] [DEFAULT default_value] [AUTO_INCREMENT]
[PRIMARY KEY] [reference_definition]
or PRIMARY KEY (index_col_name,...)
or KEY [index_name] (index_col_name,...)
or INDEX [index_name] (index_col_name,...)
or UNIQUE [INDEX] [index_name] (index_col_name,...)
or [CONSTRAINT symbol] FOREIGN KEY [index_name] (index_col_name,...)
[reference_definition]
```

type:

```
TINYINT[(length)] [UNSIGNED] [ZEROFILL]
or SMALLINT[(length)] [UNSIGNED] [ZEROFILL]
or MEDIUMINT[(length)] [UNSIGNED] [ZEROFILL]
or INT[(length)] [UNSIGNED] [ZEROFILL]
or INTEGER[(length)] [UNSIGNED] [ZEROFILL]
or BIGINT[(length)] [UNSIGNED] [ZEROFILL]
or REAL[(length,decimals)] [UNSIGNED] [ZEROFILL]
or DOUBLE[(length,decimals)] [UNSIGNED] [ZEROFILL]
or FLOAT[(length,decimals)] [UNSIGNED] [ZEROFILL]
```

```

or    DECIMAL(length,decimals) [UNSIGNED] [ZEROFILL]
or    NUMERIC(length,decimals) [UNSIGNED] [ZEROFILL]
or    CHAR(length) [BINARY]
or    VARCHAR(length) [BINARY]
or    DATE
or    TIME
or    TIMESTAMP
or    DATETIME
or    TINYBLOB
or    BLOB
or    MEDIUMBLOB
or    LONGBLOB
or    TINYTEXT
or    TEXT
or    MEDIUMTEXT
or    LONGTEXT
or    ENUM(value1,value2,value3,...)
or    SET(value1,value2,value3,...)

```

```

index_col_name:
    col_name [(length)]

```

```

reference_definition:
    REFERENCES tbl_name [(index_col_name,...)]
                [MATCH FULL | MATCH PARTIAL]
                [ON DELETE reference_option]
                [ON UPDATE reference_option]

```

```

reference_option:
    RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT

```

```

mysql> CREATE TABLE Person(
        Name VARCHAR(25)
    );

```

Beispiel 21: Erzeugung einer Tabelle

Die Anweisung aus [Beispiel 21](#) stellt den einfachsten Fall ein Tabellenerzeugungsanweisung dar. Es wird die Tabelle Person, die mit Name nur über eine Spalte verfügt erzeugt. Eine „kleinere“ Fassung ist nicht möglich, da spaltenlose Tabellen nicht erstellt werden können.

Informationen über eine angelegte Tabelle können durch den Befehl DESCRIBE gefolgt vom Namen der abzufragenden Tabelle erlangt werden:

```

mysql> DESCRIBE Person;
+-----+-----+-----+-----+-----+-----+-----+
| Field | Type          | Collation          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+-----+
| Name  | varchar(25)   | latin1_swedish_ci | YES  |     | NULL    |      |
+-----+-----+-----+-----+-----+-----+-----+

```

Beispiel 22: Ermittlung von Tabelleninformation

Im Beispiel werden die zuvor festgelegten Daten wie Spaltenname (Name) und Datentyp (VARCHAR (25)) ermittelt, sowie die Belegung einiger Vorgabewerte (u.a. NULL und Default).

Durch Angabe des Schlüsselwortes TEMPORARY können Tabellen erstellt werden, die während der Arbeit mit der Datenbank den herkömmlichen gleichgestellt behandelt werden, jedoch dem Ende der Datenbankverbindung automatisch inklusive aller darin abgelegten Daten aus der Datenbank entfernt werden.

```
mysql> CREATE TEMPORARY TABLE Person2(
        Name VARCHAR(25)
    );

--Verbindungsende
--Aufbau einer neuen Verbindung

mysql> DESCRIBE Person2;
ERROR 1146: Table 'SQLTest.Person2' doesn't exist
```

Beispiel 23: Erzeugung einer temporären Tabelle

Wird die Datenbankverbindung getrennt und neu aufgebaut, so ist die zuvor temporär erstellte Tabelle Person2 nicht mehr vorhanden und zugreifbar. Das DESCRIBE-Kommando liefert daher einen Fehler.

Wie bereits in [Beispiel 21](#) gezeigt muß jede Spalte einer Tabelle einen Datentyp besitzen. Dieser definiert die zugelassenen Wertbelegungen und wird durch das DBMS bei jeder schreibenden Operation (d.h. Einfügen, Ändern und Leeren) geprüft. Wird versucht ein ungültiger Wert zu setzen, so erfolgt eine Fehlermeldung und die Ablehnung des Eintrags- oder Änderungswunsches. Im Falle eines zeichenkettenwertigen Datentyps (z. B. VARCHAR) erfolgt keine Fehlermeldung, sondern die Werte werden nur abgeschnitten eingefügt.

```
mysql> CREATE TABLE Person(
        Name VARCHAR(25)
    );

mysql> INSERT INTO Person values("Max Mustermann");
--ok

mysql> INSERT INTO Person values("Franz Obermüller-Hinterhuber-Niedermayer");
--keine Fehlermeldung
--Wert wird jedoch nur abgeschnitten eingefügt:

mysql> SELECT * FROM Person;
+-----+
| Name                |
+-----+
| Max Mustermann      |
| Franz Obermüller-Hinterhu |
+-----+
```

Beispiel 24: Auswirkung von Datentypen I

[Beispiel 24](#) zeigt die Auswirkung eines gesetzten Datentypen VARCHAR. Werden, wie im Beispiel Werte eingefügt, die die zulässige maximale Zeichenkettenlänge überschreiten, so werden die überzähligen Zeichen ohne Fehlermeldung abgeschnitten.

```
mysql> CREATE TABLE Person(GebDat date);

mysql> INSERT INTO Person values ('1970-12-12');
--ok

mysql> insert into Person values ('1970-19-42');
--offensichtlich falsch
--Übernommener Wert: 0000-00-00
```

Beispiel 25: Auswirkung von Datentypen II

Im [Beispiel 25](#) wird ein offensichtlich ungültiges Datum eingefügt. Die Datenbank übernimmt jedoch nicht diesen falschen Wert sondern setzt den Vorgabewert von 0000-00-00. Dasselbe geschieht auch bei numerischen Datentypen (etwa: INTEGER) wenn versucht wird sie mit einer Zeichenkette zu belegen.

MySQL unterstützt drei Datentypklassen:

- Numerischdatentypen zur Darstellung von Zahlen.
- Zeichenkettendatentypen zur Darstellung von Texten und Binärdaten.
- Datumsdatentypen zur Darstellung von Uhrzeit- und Datumsformaten.

Die derzeit durch MySQL angebotenen Datentypen sind in der nachfolgenden Tabelle zusammengestellt:

Typname	Beispiel	Bemerkung
Numerische Datentypen		
TINYINT [UNSIGNED]	(Vorzeichenbehaftete) Ganzzahl der Breite acht Bit.	$(2^7, \dots, -1)$ 0, 1, ..., 2^7-1 , ($2^7, \dots, 2^8-1$) (-128, ..., -1,) 0, 1, ..., 127, (128, ..., 255)
BOOL	Boole'scher Wahrheitswert.	Zugelassene Belegungen 0 und 1. Synonym für TINYINT (1)
SMALLINT [UNSIGNED]	(Vorzeichenbehaftete) Ganzzahl der Breite 16 Bit.	$(2^{15}, \dots, -1)$ 0, 1, ..., $2^{15}-1$, ($2^{15}, \dots, 2^{16}-1$) (-32.768, ..., -1,) 0, 1, ..., 32.767, (32.768, ..., 65.535)
MEDIUMINT [UNSIGNED]	(Vorzeichenbehaftete) Ganzzahl der Breite 24 Bit.	$(2^{23}, \dots, -1)$ 0, 1, ..., $2^{23}-1$, ($2^{23}, \dots, 2^{24}-1$) (-8.388.608, ..., -1) 0, 1, ..., 8.388.607, (8.388.608, ..., 16.777.215)
INT [UNSIGNED]	(Vorzeichenbehaftete) Ganzzahl der Breite 32 Bit.	$(2^{31}, \dots, -1)$ 0, 1, ..., $2^{31}-1$, ($2^{31}, \dots, 2^{32}-1$) (-2.147.483.648, ..., -1) 0, 1, ..., 2.147.483.647, (2.147.483.648, ..., 4.294.967.295)
INTEGER [UNSIGNED]		Synonym für INT
BIGINT [UNSIGNED]	(Vorzeichenbehaftete) Ganzzahl der Breite 64 Bit.	$(2^{63}, \dots, -1)$ 0, 1, ..., $2^{63}-1$, ($2^{63}, \dots, 2^{64}-1$) (-9.223.372.036.854.774.808, ..., -1) 0, 1, ..., 9.223.372.036.854.774.807, (9.223.372.036.854.774.808, ..., 18.446.744.073.709.551.615)
FLOAT [UNSIGNED]	(Vorzeichenbehaftete) Fließkommazahl der Breite 32 Bit, gemäß dem Standard IEEE-754	$(-(2-2^{-23})^{127}, \dots, (2-2^{-23})^{127})$ -3,402... $\cdot 10^{38}$, ..., -1,175... $\cdot 10^{-38}$ und 1,175... $\cdot 10^{-38}$... 3,402... $\cdot 10^{38}$
DOUBLE [UNSIGNED]	(Vorzeichenbehaftete) Fließkommazahl der Breite 64 Bit, gemäß dem Standard IEEE-754	$(-(2-2^{-52})^{1023}, \dots, (2-2^{-52})^{1023})$ -1,797... $\cdot 10^{-308}$, ..., -2,225... $\cdot 10^{308}$ und 1,797... $\cdot 10^{308}$, ..., 2,225... $\cdot 10^{308}$
REAL [UNSIGNED]		Synonym für DOUBLE
DECIMAL [(Genauigkeit, Nachkommastellen)]	Festkommazahl beliebiger Genauigkeit	DECIMAL(9,2) liefert eine Dezimalzahl mit neun Gesamtziffern, davon zwei Nachkommastellen.
DEC		Synonym für DECIMAL
NUMERIC		Synonym für DECIMAL
Zeichenkettendatentypen		
CHAR [Länge]	Textfeld konstanter Länge bis zu 255 Zeichen, das immer die angegebene Anzahl Speicherstellen benötigt.	Dies ist ein Test0x20;0x20;0x20;
CHARACTER		Synonym zu CHAR
NCHAR		Synonym zu CHAR
NATIONAL CHARACTER		Synonym zu CHAR
CHARACTER VARYING		Synonym zu VARCHAR
NATIONAL VARCHAR		Synonym zu VARCHAR
VARCHAR [Länge]	Textfeld variabler Länge. Überzählige Leerzeichen am Ende einer Zeichenkette werden vor dem Abspeichern entfernt.	Dies ist ein Test
TINYTEXT	Textfeld variabler Länge, bis zu 2^8-1 (=255) Zeichen.	... etwas mehr Text
TEXT	Textfeld variabler Länge, bis zu $2^{16}-1$ (=65.535) Zeichen.	Sehr viel ... Text hier ...
MEDIUMTEXT	Textfeld variabler Länge, bis zu $2^{24}-1$ (=16.777.215) Zeichen.	... etwas mehr Text hier ...
LONGTEXT	Textfeld variabler Länge, bis zu $2^{32}-1$ (=2.294.967.295) Zeichen.	... noch mehr Text hier ...
TINYBLOB	Binäre Form von TINYTEXT.	
MEDIUMBLOB	Binäre Form von MEDIUMTEXT	

	BLOB	Binäre Form von TEXT
Datumstypen		
DATE	Datum in ISO-8601-Schreibweise (JJJJ-MM-TT)	2004-01-08
TIME	Uhrzeit gemäß ISO 8601 (hh:mm:ss)	07:45:00
DATETIME	Datum und Uhrzeit gemäß ISO 8601 (JJJJ-MM-TT hh:mm:ss)	2005-05-27 07:45:00
TIMESTAMP	Sekundengenauer Zeitpunkt zwischen 1970-01-01 und 2037-12-31	Anwenderdarstellung: 2004-01-08 22:57:49
YEAR	Vierstellige Jahreszahl	2004
Komplexe Datentypen		
ENUM	Aufzählungstyp mit bis zu 65.535 Elementen	
SET	Menge mit bis zu 64 Elementen	

Jede Spalte einer Relation muß mit genau einem Datentyp der oben dargestellten Liste versehen werden. Nachfolgend sind einige Definitionen und Besonderheiten zusammengestellt:

```
mysql> CREATE TABLE test(wenigText CHAR(300));
--zulässige Grenze für CHAR überschritten ...

mysql> DESCRIBE test;
+-----+-----+-----+-----+-----+-----+-----+
| Field      | Type | Collation          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+-----+
| wenigText  | text | latin1_swedish_ci | YES  |     | NULL    |      |
+-----+-----+-----+-----+-----+-----+-----+
```

Beispiel 26: Auswirkung von Datentypen III

[Beispiel 26](#) zeigt die automatische Konversion des Datentypen CHAR in der Datenbank in TEXT sobald bereits bei der Anlage die zulässige Größenbegrenzung für CHAR überschritten wird. Werden zur Laufzeit Texte größer als die im CREATE TABLE-Ausdruck angegebene Maximalkapazität abgespeichert, so werden alle überzähligen Zeichen abgeschnitten.

```
mysql> CREATE TABLE test(ts timestamp, x VARCHAR(10));
Query OK, 0 rows affected (0.35 sec)

mysql> INSERT INTO test values(null, "abc");
Query OK, 1 row affected (0.06 sec)

mysql> INSERT INTO test values(null, "def");
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO test values(null, "abc");
Query OK, 1 row affected (0.06 sec)

mysql> SELECT * FROM test;
+-----+-----+
| ts              | x    |
+-----+-----+
| 2003-05-26 23:25:51 | abc  |
| 2003-05-26 23:26:13 | def  |
| 2003-05-26 23:26:28 | abc  |
+-----+-----+
3 rows in set (0.00 sec)

mysql> UPDATE test SET x="xyz" WHERE x="abc";
Query OK, 2 rows affected (0.05 sec)
Rows matched: 2  Changed: 2  Warnings: 0

mysql> SELECT * FROM test;
+-----+-----+
| ts              | x    |
+-----+-----+
| 2003-05-26 23:27:10 | xyz  |
| 2003-05-26 23:26:13 | def  |
+-----+-----+
```



```
| 2003-05-26 23:27:10 | xyz |
+-----+-----+
3 rows in set (0.00 sec)
```

Beispiel 27: Auswirkung von Datentypen IV

Das Beispiel zeigt die Nutzung des Typs `TIMESTAMP`. Spalten dieses Typs werden automatisch durch das DBMS mit Werten versorgt werden. Daher ist die Belegung mit `NULL` bei Einfügung der drei Zeichenketten wirkungslos.

Überdies wird der Wert jeder `TIMESTAMP`-typisierten Spalte (im Beispiel: `ts`) bei jedem Schreibvorgang aktualisiert. Dies zeigt die nochmalige Ausgabe der Tabelleninhalte nach Aktualisierung der beiden Tupel, die für das Attribut `x` den Wert `abc` aufweisen.

```
mysql> create table Ampel(farbe enum('rot','gelb','gruen'));
Query OK, 0 rows affected (0.07 sec)

mysql> INSERT INTO Ampel VALUES('rot');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT into Ampel VALUES('blau');
Query OK, 1 row affected (0.04 sec)

mysql> SELECT * FROM Ampel;
+-----+
| farbe |
+-----+
| rot   |
|      |
+-----+

mysql> INSERT INTO Ampel Values(2);
Query OK, 1 row affected (0.05 sec)

mysql> SELECT * FROM Ampel;
+-----+
| farbe |
+-----+
| rot   |
|      |
| gelb  |
+-----+
3 rows in set (0.00 sec)
```

Beispiel 28: Auswirkung von Datentypen V

Das [Beispiel 28](#) zeigt die Nutzung eines Aufzählungstypen.

Er erlaubt ausschließlich das Einfügen der vordefinierten Werte und legt für alle ungültigen Belegungen (im Beispiel: `blau`) die leeren Zeichenkette ab.

Die Werte können dabei wie in der Aufzählung definiert oder durch ihre Indexposition (beginnend ab 1) gespeichert werden.

```
mysql> CREATE TABLE test(x SET("a","b","c","d"));
Query OK, 0 rows affected (0.07 sec)

mysql> INSERT INTO test VALUES("a");
Query OK, 1 row affected (0.07 sec)

mysql> INSERT INTO test values("a,b");
Query OK, 1 row affected (0.04 sec)

mysql> INSERT INTO test values("a,c");
Query OK, 1 row affected (0.07 sec)

mysql> INSERT INTO test values("b,c");
Query OK, 1 row affected (0.05 sec)

mysql> SELECT * FROM test;
+-----+
```

```

| x |
+-----+
| a |
| a,b |
| a,c |
| b,c |
+-----+
4 rows in set (0.00 sec)

```

```
mysql> select * FROM test WHERE x & 1;
```

```

+-----+
| x |
+-----+
| a |
| a,b |
| a,c |
+-----+
3 rows in set (0.00 sec)

```

--liefert alle Tupel, die das zweite Mengenelement ("a") enthalten

Beispiel 29: Auswirkung von Datentypen VI

Das Beispiel zeigt die Definition eines mengenwertigen Datentyps, d.h. einer Tabellenspalte, die mehr als einen Wert aufnehmen kann sowie die Abfragemöglichkeiten dafür.

Hinweis: Dieser Datentyp führt bereits in die [NF2](#)-Datenstrukturen über und wird daher nicht im Rahmen des Entwurfsprozesses im konzeptuellen Schema verwendet.

Ergänzend zur Datentypangabe können für jede Spalte weitere einschränkende Angaben zur Spezifikation der erlaubten Werte getroffen werden.

NOT NULL legt hierbei fest, daß ein Tupel für eine Spalte zwingend einen von NULL verschiedenen Wert besitzen muß.

```
mysql> CREATE TABLE Person(Name VARCHAR(20), PersAuswNr INT NOT NULL);
Query OK, 0 rows affected (0.08 sec)
```

```
mysql> INSERT INTO Person VALUES('Max Obermüller', '123456789');
Query OK, 1 row affected (0.11 sec)
```

```
mysql> INSERT INTO Person VALUES('Xaver Hinterhuber', NULL);
ERROR 1048: Column 'PersAuswNr' cannot be null
```

```
mysql> select * from Person;
+-----+-----+
| Name          | PersAuswNr |
+-----+-----+
| Max Obermüller | 123456789 |
+-----+-----+
1 row in set (0.00 sec)

```

Beispiel 30: Definition einer Spalte als NOT NULL

Das Beispiel zeigt eine Tabelle, bei der das Attribut PersAuswNr als NOT NULL definiert wurde. Einfüge- oder Aktualisierungsversuche, die zu Nullwerten dieses Attributs führen würden, werden durch das DMBS unterbunden.

Alternativ dazu gestattet die Angabe von NULL die Existenz von Nullwerten in der Tabelle. Diese Definition ist optional und wird bei fehlender Angabe automatisch als Vorgabe gesetzt. Das Beispiel zeigt die Definition der Spalte Autofahrer als NULL, die neben den beiden vorgegebenen Werten auch keinen Wert enthalten darf.

```
mysql> CREATE TABLE Person(
  -> Name VARCHAR(20),
  -> PersAuswNr INT NOT NULL,
  -> Autofahrer ENUM('J','N') NULL);
Query OK, 0 rows affected (0.07 sec)

mysql> INSERT INTO Person VALUES('Max Obermüller', '123456789', NULL);
Query OK, 1 row affected (0.12 sec)

mysql> INSERT INTO Person VALUES('Xaver Hinterhuber', '234567891', 'J');
Query OK, 1 row affected (0.04 sec)

mysql> select * from Person;
+-----+-----+-----+
| Name          | PersAuswNr | Autofahrer |
+-----+-----+-----+
| Max Obermüller | 123456789  | NULL       |
| Xaver Hinterhuber | 234567891  | J          |
+-----+-----+-----+
3 rows in set (0.01 sec)
```

Beispiel 31: Definition einer Spalte als NULL

Die Angabe der Klausel `DEFAULT VALUE` gestattet es einen Vorgabewert zu definieren, der gesetzt wird wenn kein Wert für eine Spalte angegeben wird.

Dies ersetzt jedoch nicht automatisch die Möglichkeit des Auftretens von Nullwerten innerhalb einer Spalte. Diese können auch weiterhin auftreten, sofern sie explizit eingefügt werden.

Die unterschiedlichen Wirkungsweisen zeigt [Beispiel 32](#). Dort findet sich die Spalte `Autofahrer` (vorgabegemäß, da keine andere Angabe erfolgte) als nullwertfähig mit Vorgabewert J, sowie die Spalte `Hundebesitzer`, die mit Vorgabe N und als nicht nullwertfähig deklariert wurde.

```
mysql> CREATE TABLE Person(
  -> Name VARCHAR(20) NOT NULL,
  -> Autofahrer ENUM('J','N') DEFAULT 'J',
  -> Hundebesitzer ENUM('J','N') NOT NULL DEFAULT 'N'
  -> );
Query OK, 0 rows affected (0.07 sec)

mysql> INSERT INTO Person VALUES('Xaver Obermüller', 'J', 'J');
Query OK, 1 row affected (0.10 sec)

mysql> INSERT INTO Person VALUES('Max Hinterhuber', NULL, 'N');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Person (Name) VALUES('Schorsch Huber');
Query OK, 1 row affected (0.05 sec)

mysql> SELECT * FROM Person;
+-----+-----+-----+
| Name          | Autofahrer | Hundebesitzer |
+-----+-----+-----+
| Xaver Obermüller | J          | J              |
| Max Hinterhuber  | NULL       | N              |
| Schorsch Huber   | J          | N              |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

Beispiel 32: Definition einer Spalte mit Vorgabewerten

Ist die Auszeichnung einer einzelnen Spalte als [Primärschlüssel](#) gewünscht, so kann der Definition (`PRIMARY`) `KEY` nachgestellt werden um dies zu erreichen.

Die Definition eines Primärschlüssels impliziert den Zwang in jedem Tupel einen von NULL verschiedenen eindeutigen Wert dafür abspeichern zu müssen. Primärschlüsselspalten sind damit immer auch `NOT NULL`.

Zusätzlich kann für jede Tabelle höchstens ein Primärschlüsselattribut angegeben werden.

Hinweis: Aus mehr als einer Spalte zusammengesetzte Primärschlüssel können nicht durch diese Syntax gebildet werden.

```
mysql> CREATE TABLE Person(Name VARCHAR(20) PRIMARY KEY, Adresse VARCHAR(50));
Query OK, 0 rows affected (0.06 sec)

mysql> INSERT INTO Person VALUES('Xaver Obermüller', 'Dorfstr. 12');
Query OK, 1 row affected (0.10 sec)

mysql> INSERT INTO Person VALUES('Hans Hintermeier', 'Dorfstr. 13');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Person (Name) VALUES ('Schorsch Huber');
Query OK, 1 row affected (0.06 sec)

mysql> select * from Person;
+-----+-----+
| Name          | Adresse      |
+-----+-----+
| Hans Hintermeier | Dorfstr. 13  |
| Schorsch Huber   | NULL         |
| Xaver Obermüller | Dorfstr. 12  |
+-----+-----+
4 rows in set (0.00 sec)

mysql> INSERT INTO Person VALUES(NULL, 'Hauptstr. 11');
ERROR 1048: Column 'Name' cannot be null
```

Beispiel 33: Definition eines Primärschlüssels

Zur Erstellung eines zusammengesetzten Primärschlüssels kann nicht das nachgestellte Schlüsselwort `PRIMARY KEY` verwendet werden, da seine wiederholte Angabe mehrdeutig wäre. Für diesen Fall muß eine gesondert `PRIMARY KEY`-Definition in den `CREATE TABLE`-Ausdruck aufgenommen werden:

```
CREATE TABLE DEPT_LOCATIONS(
    DNUMBER INTEGER(1) NOT NULL,
    DLOCATION VARCHAR(20) NOT NULL,
    PRIMARY KEY (DNUMBER, DLOCATION));
```

Beispiel 34: Definition eines zusammengesetzten Primärschlüssels

Das Beispiel zeigt verschiedene Einfügeoperationen. Bemerkenswert ist die letzte, die das Primärschlüsselattribut leer läßt. Die hierbei erfolgende Belegung mit der leeren Zeichenkette (nicht `NULL`!) ist ein gültiger Eintrag im Sinne des angegebenen Datentypen `VARCHAR` und der Restriktion keine Belegung mit `NULL` vorzunehmen.

Soll ein nicht-sprechender Schlüssel (z.B. eine einfache Zählnummer) zur Identifikation genutzt werden, so kann diese durch das DBMS automatisiert bereitgestellt werden.

```
mysql> CREATE TABLE Person(
    -> LfdNr Int AUTO_INCREMENT PRIMARY KEY,
    -> Name VARCHAR(20) NOT NULL);
Query OK, 0 rows affected (0.07 sec)

mysql> INSERT INTO Person VALUES(1, 'Max Obermüller');
Query OK, 1 row affected (0.09 sec)

mysql> INSERT INTO Person VALUES(3, 'Schorsch Hinterhuber');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Person VALUES(3, 'Xaver Mayer');
ERROR 1062: Duplicate entry '3' for key 1

mysql> SELECT * FROM Person;
+-----+-----+
| LfdNr | Name          |
+-----+-----+
| 1     | Max Obermüller |
| 3     | Schorsch Hinterhuber |
```

```

+-----+-----+
2 rows in set (0.00 sec)

mysql> INSERT INTO Person (Name) VALUES('Xaver Mayer');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Person (Name) VALUES('Hans Huber');
Query OK, 1 row affected (0.04 sec)

mysql> select * from Person;
+-----+-----+
| LfdNr | Name                |
+-----+-----+
|      1 | Max Obermüller     |
|      2 | Xaver Mayer         |
|      3 | Schorsch Hinterhuber |
|      4 | Hans Huber          |
+-----+-----+
4 rows in set (0.00 sec)

```

Beispiel 35: Definition eines automatisch befüllten Primärschlüssels

Im [Beispiel 35](#) wird der Wert der Spalte LfdNr, sofern nicht durch den Anwender explizit angegeben, automatisch ermittelt und eingefügt.

Zur Zugriffsbeschleunigung dienende Indexstrukturen können bereits zum Tabellenerstellungszeitpunkt durch den Anwender angegeben werden. Diese werden jedoch nicht der Spaltendefinition als nachgestellt, sondern bilden einen eigenen Eintrag innerhalb der Tabellendefinition.

Ein Index kann gleichzeitig eine oder mehrere Spalten umfassen. Beispiel [Beispiel 36](#) zeigt dies:

```

mysql> CREATE TABLE Person(
  -> Name VARCHAR(20) PRIMARY KEY,
  -> GebDat DATE, Str_HsNr VARCHAR(20),
  -> PLZ CHAR(5),
  -> Ort VARCHAR(50),
  -> INDEX GebDatIdx (GebDat),
  -> INDEX AdresseIndex (Str_HsNr, PLZ, Ort));
Query OK, 0 rows affected (0.07 sec)

```

Beispiel 36: Definition von Indexen

Als beschränkende Verschärfung, die sich auch positiv auf die Zugriffsgeschwindigkeit auswirkt, kann ein Index als `UNIQUE INDEX` definiert werden. Er darf dann ausschließlich eindeutige Werte oder Wertkombinationen aufnehmen.

Erzeugung von Fremdschlüsselbeziehungen

Die Erzeugung von Fremdschlüsselbeziehungen ist das integrale Element zur Wahrung der referentiellen Integrität. Fremdschlüsselbeziehungen können bereits zum Erstellungszeitpunkt einer Tabelle angegeben werden wie [Beispiel 37](#) zeigt oder nachträglich durch einen `ALTER TABLE`-Ausdruck hinzugefügt werden wie durch [Beispiel 38](#) gezeigt.

```

CREATE TABLE DEPARTMENT(
  DNAME VARCHAR(20) NOT NULL,
  DNUMBER INTEGER(1) PRIMARY KEY,
  MGRSSN INTEGER(9),
  MGRSTARTDATE DATE);

CREATE TABLE EMPLOYEE(
  FNAME VARCHAR(10) NOT NULL,
  MINIT VARCHAR(1),
  LNAME VARCHAR(10) NOT NULL,
  SSN INTEGER(9) PRIMARY KEY,
  BDATE DATE,
  ADDRESS VARCHAR(30),
  SEX ENUM('M', 'F'),

```

```
SALARY REAL(7,2) UNSIGNED,
SUPERSSN INTEGER(9),
DNO INTEGER(1) REFERENCES DEPARTMENT(DNUMBER),
INDEX DNO_IDX (DNO));
```

Beispiel 37: Erzeugung von Fremdschlüsselbeziehungen zum Tabellenerstellungszeitpunkt

Das Beispiel erzeugt neben der angestrebten Fremdschlüsselbeziehungen zwischen dem Attribut DNO der Tabelle EMPLOYEE und dem Primärschlüssel DNUMBER in DEPARTMENT einen Index auf den Fremdschlüssel innerhalb der Tabelle EMPLOYEE.

Dies ist für einige Datenbankmanagementsysteme (darunter MySQL) notwendig, um die Zugriffe auf den Fremdschlüssel zu beschleunigen.

Das nachfolgende Beispiel liefert dasselbe Ergebnis, jedoch unter nachträglicher (d.h. nach dem Erstellungszeitpunkt der Tabellen) Fremdschlüsselerzeugung:

```
CREATE TABLE DEPARTMENT(
    DNAME VARCHAR(20) NOT NULL,
    DNUMBER INTEGER(1) PRIMARY KEY,
    MGRSSN INTEGER(9),
    MGRSTARTDATE DATE);

CREATE TABLE EMPLOYEE(
    FNAME VARCHAR(10) NOT NULL,
    MINIT VARCHAR(1),
    LNAME VARCHAR(10) NOT NULL,
    SSN INTEGER(9) PRIMARY KEY,
    BDATE DATE,
    ADDRESS VARCHAR(30),
    SEX ENUM('M', 'F'),
    SALARY REAL(7,2) UNSIGNED,
    SUPERSSN INTEGER(9),
    DNO INTEGER(1));

ALTER TABLE EMPLOYEE ADD INDEX DNO_IDX (DNO);
ALTER TABLE EMPLOYEE ADD CONSTRAINT DNO_FK FOREIGN KEY (DNO) REFERENCES DEPARTMENT
(DNUMBER);
```

Beispiel 38: Nachträgliche Erzeugung von Fremdschlüsselbeziehungen

3.3 Der Anfrageteil von SQL

Anfragen zur Ermittlung von Datenbankinhalten stellen den eigentlichen Sprachkern von SQL und zweifellos den in der Praxis bedeutsamsten Anteil der Sprache dar.

Die gesamte Mächtigkeit des Anfrageteils von SQL erschließt sich durch das Schlüsselwort SELECT. Es gestattet Anfragen theoretisch unbegrenzter Komplexität in einer uniformen und leicht zu behaltenden Syntax zu formulieren, deren Mächtigkeit von einfachsten Anfragen bis zu aufwendigen Auswertungen reicht.

Anfragen von Datenbankinhalten

Die SQL-Anweisung SELECT dient der Abfrage von in einer Datenbank abgelegten Inhalten. Sie benötigt Wissen über die angelegten Tabellen sowie deren Struktur hinsichtlich Spalten und deren Typen.

Alle nachfolgenden Beispielanfragen beziehen sich, sofern nicht anders angegeben auf die [Demodatenbank](http://www.jeckle.de/vorlesung/datenbanken/script.html).

Die vereinfachte Syntax der SELECT-Anweisung lautet:

```
SELECT [ALL|DISTINCT] select_item,...
FROM table_specification,...
[WHERE search_condition]
[GROUP BY grouping_column,...]
[HAVING search_condition]
```

[ORDER BY sort_specification,...]

```
SELECT FNAME
FROM EMPLOYEE;
```

Beispiel 39: Einfache Anfrage

Die Anfrage liefert die Inhalte der Spalte FNAME aller in der Tabelle EMPLOYEE abgelegten Tupel. Die Werte werden in keiner vorgegebenen Reihenfolge ausgegeben, d.h. eine etwaige Sortierung ist zufallsbedingt und kann durch DBMS interne Reorganisationsprozesse zerstört werden.

Durch diese Anfrage wird als Resultat eine nicht in der Datenbank abgelegte Tabelle erzeugt, welche nur die im SELECT-Ausdruck angegebenen Spalten enthält. Die Ergebnistabelle stimmt zwar in Tupelanzahl mit der Ursprungstabelle überein, blendet jedoch einzelne Attribute aus. Dieser Vorgang wird als *Projektion* bezeichnet.

Definition 42: *Projektion*

Die Projektion blendet einzelne Spalten aus.

```
SELECT FNAME, MINIT, LNAME, SSN, BDATE, ADDRESS, SEX, SALARY, SUPERSSN, DNO
FROM EMPLOYEE;
```

Beispiel 40: Anfrage aller Spalten einer Tabelle

Die Abfrage aus [Beispiel 40](#) liefert die Wertinhalte aller Spalten (sie sind explizit nach dem Schlüsselwort SELECT angegeben) der Relation EMPLOYEE.

Als Besonderheit wird für das Attribut SUPERSSN des Mitarbeiters James E. Borg der Wert NULL ausgegeben. Diese Zeichenkette gibt an, daß für dieses Attribut der Relation kein Wert abgespeichert wurde.

Die schreibaufwendige und damit fehlerträchtige Explizierung aller Spalten einer Relation ist kaum praktikabel und überdies äußerst änderungssensitiv im Falle der Aufnahme neuer Spalten oder der Lösung Existierender.

Aus diesem Grunde kann statt des Spaltennamens ein Stern „*“ als Jokerzeichen stellvertretend für alle Spalten einer Relation angegeben werden.

[Beispiel 41](#) zeigt dies als Umschreibung der Anfrage aus [Beispiel 40](#):

```
SELECT *
FROM EMPLOYEE;
```

Beispiel 41: Anfrage aller Spalten einer Tabelle mit Jokerzeichen

Enthält die Ausgabe nicht den Primärschlüssel einer Tabelle, so kann es vorkommen, das mehrfach dieselben Werte ausgegeben werden. Dies kann durch Angabe des Schlüsselwortes DISTINCT in der SELECT-Klausel vermieden werden.

DISTINCT überschreibt das vorgegebene Verhalten (ALL) alle Einträge auszugeben.

```
SELECT DISTINCT SALARY
FROM EMPLOYEE;
```

Beispiel 42: Duplikatfreie Ausgabe aller verschiedenen Werte

[Beispiel 42](#) liefert alle verschiedenen Werteinträge der Spalte SALARY duplikatfrei.

Durch Angabe mehrerer Einträge in der FROM-Klausel können Inhalte aus verschiedenen Tabellen innerhalb einer Anfrage extrahiert werden:

```
SELECT DNAME, PNUMBER
FROM DEPARTMENT, PROJECT;
```

Beispiel 43: Anfrage auf zwei Tabellen

Die Anfrage bildet das [kartesische Produkt](#) der beiden angefragten Tabellen.

Aliasbildung

Bei Anfragen über mehrere Tabellen kann es zu Problemen hinsichtlich der Eindeutigkeit der Spaltenbezeichner kommen. So würde die Anfrage aus [Beispiel 44](#) nicht die für ESSN abgelegten Werte liefern, sondern den Fehler ERROR 1052: Column: 'ESSN' in field list is ambiguous, da in jeder der beiden in der Anfrage berücksichtigten Tabellen eine Spalte mit ESSN benannt ist.

```
SELECT ESSN, ESSN
FROM WORKS_ON, DEPENDENT;
```

Beispiel 44: Fehlerhafte Anfrage auf zwei Tabellen

Als Lösung bietet SQL die Möglichkeit den Spaltennamen zusätzlich durch Voranstellung des Namens der die Spalte beherbergenden Tabelle zu qualifizieren um die erforderliche Eindeutigkeit herzustellen:

```
SELECT WORKS_ON.ESSN, DEPENDENT.ESSN
FROM WORKS_ON, DEPENDENT;
```

Beispiel 45: Lösung des Mehrdeutigkeitsproblems bei Anfrage auf zwei Tabellen

Unter Nutzung der Möglichkeit Alternativnamen für Tabellen, sog. *Aliasnamen*, anzugeben ergibt sich eine in der Schreibung kompaktere Umsetzung:

```
SELECT w.ESSN, d.ESSN
FROM WORKS_ON AS w, DEPENDENT AS d;
```

Beispiel 46: Lösung des Mehrdeutigkeitsproblems bei Anfrage auf zwei Tabellen

Gleichzeitig kann die Aliasbildung eingesetzt werden, um die Benennung der Spalten bei der Ausgabe zu modifizieren. Auf diesem Wege können wenig sprechende Namen oder Doppelbenennungen umgangen werden.

```
SELECT w.ESSN AS "Mitarbeiter Sozialversicherungsnummer",
d.ESSN AS "Sozialversicherungsnummer des Verwandten"
FROM WORKS_ON AS w, DEPENDENT AS d;
```

Beispiel 47: Umbenennung von Ausgabespalten

Berechnete Ausgaben

Durch die Angabe einfacher arithmetischer Formeln in der SELECT-Klausel können vor ihrer Ausgabe Berechnungen auf den Werten aus der Datenbank angestellt werden.

```
SELECT FNAME, SALARY*12 as Jahreseinkommen
FROM EMPLOYEE;
```

Beispiel 48: Berechnungen I

Beschränkung der Ergebnismenge

Die bisher betrachteten Anfrageformen lieferten immer die gesamten Inhalte der betrachteten Tabellen. Durch Angabe einer einschränkenden Bedingung innerhalb der WHERE-Klausel einer SELECT-Anweisung können die Tabelleninhalte hinsichtlich einer Bedingung gefiltert werden.

[Beispiel 49](#) liefert nur die abgespeicherten Werte für Geburtsdatum (BDATE) und Adresse (ADDRESS) derjenigen Mitarbeiter, deren Vornamen (FNAME) *John* ist.


```
SELECT BDATE, ADDRESS
FROM EMPLOYEE
WHERE FNAME="John";
```

Beispiel 49: Einschränkung der Anfrage

Durch die Filterung der Ergebnistupel werden alle diejenigen Datenbankeinträge, welche die getroffene Bedingung nicht erfüllen ausgeblendet. Das Ergebnis kann (sofern SELECT * ... gewählt wurde) zwar in der Anzahl Spalten mit der Ursprungsrelation übereinstimmen, wird dies jedoch typischerweise (d.h. außer im Falle, daß alle Tupel der Relation die formulierte Bedingung erfüllen) nicht tun.

Dieser Vorgang wird als *Selektion* bezeichnet.

Definition 43: Selektion

Die Selektion blendet einzelne Werteinträge aus..

Als relationale Operatoren für Vergleichstests zwischen zwei (möglicherweise einelementigen) Mengen stehen zur Verfügung:

Operator	Funktion	Bemerkung
=	Gleichheitstest	
<>	Test auf Ungleichheit	Teilweise (auch in MySQL!) ist der Standardoperator durch != ersetzt
<	Test auf kleiner	Liefert bei Zeichenkettendatentypen alle lexikalisch „kleineren“, d.h. diejenigen die in der alphabetischen Sortierung früher auftreten. Ebenso liefert der Operator bei der Anwendung auf Datumsdatentypen die kalendarisch früheren.
<=	Kleiner oder gleich	
>	Größer	
>=	Größer oder gleich	
IS NULL	Testet ob eine Spalte NULL enthält	
IS NOT NULL	Testet ob eine Spalte nicht NULL enthält	
BETWEEN	Testet ob ein Wert in vorgegebenen Grenzen liegt	
IN	Testet ob ein Wert innerhalb einer vorgegebenen Menge liegt	

Neben den einfachen Vergleichsoperationen können durch den LIKE-Operator unscharfe musterbasierte Suchen ausgedrückt werden.

Die Musterausdrücke werden dabei aus den tatsächlich in der Ergebnismenge erwarteten Zeichen ergänzt um Metazeichen mit besonderer Bedeutung zusammengesetzt. Hierbei stehen „%“ zur Stellvertretung einer (möglicherweise leeren) Menge beliebiger Zeichen und „_“ zur Stellvertretung genau eines Zeichens zur Verfügung.

```
SELECT BDATE, ADDRESS
FROM EMPLOYEE
WHERE FNAME LIKE "J%";
```

Beispiel 50: Musterbasierte Anfrage I

Die Anfrage aus [Beispiel 50](#) liefert die Werte der Spalten BDATE und ADDRESS aller Mitarbeiter deren Name (FNAME) mit einem „J“ beginnt.

Die Anfrage aus [Beispiel 51](#) beschränkt die Suche zusätzlich auf diejenigen Namen, deren vorletztes Zeichen ein „e“ ist, d.h. diejenigen Einträge für die nach dem „e“ nur noch genau ein beliebiges Zeichen auftritt.

```
SELECT BDATE, ADDRESS
FROM EMPLOYEE
WHERE FNAME LIKE "J%e_";
```

Beispiel 51: Musterbasierte Anfrage II

Die Variante aus [Beispiel 52](#) extrahiert alle Spalteninhalte der Mitarbeitertabelle deren Name aus

genau fünf Zeichen besteht.

```
SELECT *
FROM EMPLOYEE
WHERE FNAME LIKE "_____";
```

Beispiel 52: Musterbasierte Anfrage III

Kombination von Einzelbedingungen

Zur Selektion nach mehreren Bedingungen können diese mit den logischen Operationen AND, OR und NOT kombiniert werden.

Die Anfrage aus [Beispiel 53](#) liefert die Namen aller Mitarbeiter, die in „Houston“ wohnen und weniger als 50000 verdienen.

```
SELECT FNAME
FROM EMPLOYEE
WHERE ADDRESS LIKE "%Houston%" AND SALARY < 50000;
```

Beispiel 53: Kombination von Bedingungen

Mittels der Verknüpfungsoperatoren können auch einige der zuvor gezeigten Vergleichsoperatoren ausgedrückt werden.

Vergleichsoperator	Alternative Schreibweise mit Bedingungsverknüpfung
a BETWEEN b and c	(a >= b) AND (a <= c)
x IN (a, b, c)	(x = a) OR (x = b) OR (x = c)

Für die Kombinationsoperatoren gilt, aufgrund der Möglichkeit des Auftretens von NULL-Werten, die dreiwertige Logik:

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

Kombination von Anfrageergebnissen

In manchen Fällen ist es gewünscht das Ergebnis eigenständiger Anfragen zu einem Ergebnis zu kombinieren. Hierzu kann das UNION-Schlüsselwort zur Verbindung der Einzelanfragen.

```
CREATE TABLE Artikel(
    ArtNo          VARCHAR(4) PRIMARY KEY,
    Bezeichnung    VARCHAR(10) NOT NULL,
    Preis          DECIMAL(5,2) NOT NULL);

CREATE TABLE Sonderpreise(
    ArtikelNo      VARCHAR(4) PRIMARY KEY,
    Bezeichnung    VARCHAR(10) NOT NULL,
    Sonderverkaufsgrund VARCHAR(20));

INSERT INTO Artikel VALUES("2222", "Blitz Superrein", 19.99);
INSERT INTO Artikel VALUES("1111", "Wusch Superfein", 99.95);

INSERT INTO Sonderpreise VALUES("4444", "Kratzweich", "Wasserschaden");
```

```

INSERT INTO Sonderpreise VALUES("3333", "Glanz Extraweiss", NULL);

SELECT ArtNo
FROM Artikel
UNION
SELECT ArtikelNo
FROM Sonderpreise;

```

Beispiel 54: Kombination mittels UNION

Die Anfrage aus [Beispiel 54](#) erstellt zunächst zwei Tabellen und fügt einige Daten ein. Anschließend werden die Artikelnummern (Spalte ArtNo bzw. ArtikelNo) beider Tabellen angefragt und das Ergebnis mittels UNION zu einer Resultattabelle kombiniert.

Voraussetzung der Kombinierbarkeit ist die Typgleichheit der selektierten und zu vereinigenden Attribute (im Beispiel beide vom Typ VARCHAR(4)).

Implizit führt die Verwendung von UNION sowohl die Sortierung der Ergebnismenge als auch die Duplikatentfernung daraus herbei.

Verbünde

Häufig besteht der Wunsch Werte aus verschiedenen Tabellen nicht nur gemeinsam abzufragen und anzuzeigen, sondern auch inhaltlich in Beziehung zu setzen.

Die Anfrage aus [Beispiel 55](#) versucht durch Abfrage der Tabellen EMPLOYEE und DEPARTMENT die Namen der Mitarbeiter und die (in der anderen Tabelle abgelegte) Bezeichnung Abteilung zu ermitteln die sie beschäftigen.

Aufgrund der Bildung des kartesischen Produkts werden jedoch alle theoretisch möglichen Kombinationen geliefert und nicht die Untermenge der tatsächlich existierenden Paarungen.

```

SELECT FNAME, DNAME
FROM DEPARTMENT, EMPLOYEE;

```

Beispiel 55: Fehlerhafte Verbundbildung

Durch (geschickte) Nutzung der WHERE-Bedingung, die Werte aus beiden Tabellen miteinander in Beziehung setzt, gelingt jedoch die gewünschte Ermittlung:

```

SELECT FNAME, DNAME
FROM DEPARTMENT AS d, EMPLOYEE as e
WHERE d.DNUMBER = e.DNO;

```

Beispiel 56: Innerer Verbund

Das [Beispiel 56](#) liefert lediglich diejenigen Tupel, für die das in EMPLOYEE abgespeicherte DNO-Attribut einen Wert enthält, der auch in der Spalte DNUMBER der Tabelle DEPARTMENT auftritt.

Definition 44: Innerer Verbund

Ein Innerer Verbund enthält die selektierten Daten aller beteiligten Tabellen, welche die formulierte Einschränkungsbefingung erfüllen.

Der SQL-Standard gibt für diese besondere Anfrageform eine eigene Syntax vor:

```

SELECT FNAME, DNAME
FROM DEPARTMENT AS d INNER JOIN EMPLOYEE AS e
ON d.DNUMBER = e.DNO;

```

Beispiel 57: Innerer Verbund in Standardnotation

Die Bildung von Verbünden ist nicht auf die Angabe verschiedener Tabellen beschränkt, sondern kann auch durch mehrfache Bezugnahme auf dieselbe Tabelle geschehen:

```
SELECT e1.FNAME as Chef, e2.FNAME as Mitarbeiter
FROM EMPLOYEE AS e1, EMPLOYEE AS e2
WHERE e1.SSN = e2.SUPERSSN;
```

Beispiel 58: Innerer Verbund unter mehrfacher Nutzung derselben Tabelle

[Beispiel 59](#) zeigt ein Beispiel der Bildung eines inneren Verbundes unter Zugriff auf drei Tabellen. Die Anfrage liefert die Familiennamen (Tabelle EMPLOYEE) sowie die Abteilungen denen der Mitarbeiter zugeordnet ist (aus Tabelle DEPARTMENT) sowie die durch den Mitarbeiter bearbeiteten Projekte (Tabelle PROJECT).

Die Tabelle PROJECT kann jedoch nicht direkt in den Verbund einbezogen werden, da sie über keine geeigneten Attribute (d.h. Attribute die mit derselben Semantik in einer der beiden anderen Tabellen auftreten) verfügt. Daher wird zusätzlich die Tabelle WORKS_ON in die Anfrage miteinbezogen, weil sie mit dem Attribut ESSN ein Attribut bietet, welches die in EMPLOYEE enthaltene Attribut SSN als Fremdschlüssel beinhaltet. Ausgehend hiervon kann eine Bedingung unter Einbezug von PROJECT formuliert werden.

```
SELECT FNAME, DNAME, PNAME
FROM EMPLOYEE AS e, DEPARTMENT AS d, PROJECT AS p, WORKS_ON AS w
WHERE d.DNUMBER = e.DNO AND e.SSN = w.ESSN AND w.PNO = p.PNUMBER;
```

Beispiel 59: Innerer Verbund dreier Tabellen

Für Verbünde ist die Bildung durch ausschließliche Nutzung des Gleichheitsoperators innerhalb der WHERE-Klausel keineswegs zwingend, wenngleich diese sog. *Equi Joins* eine häufige Anwendungsform darstellen.

[Beispiel 1](#) zeigt einen durch Nutzung des kleiner-Operators gebildeten Verbund, der alle Abteilungen enthält, in denen ein Mitarbeiter (noch) nicht arbeitet und deren Abteilungsnummer größer ist als die Nummer der Abteilung welcher der Mitarbeiter gegenwärtig zugeordnet ist. (Mögliche semantische Deutung: Liste möglicher Beförderungen, sofern größere Abteilungsnummern einen Aufstieg codieren.)

```
SELECT FNAME, DNAME
FROM EMPLOYEE JOIN DEPARTMENT
ON DEPARTMENT.DNUMBER < EMPLOYEE.DNO;
```

Beispiel 60: Non-Equi-Join

Äußere Verbunde

Neben der Möglichkeit durch innere Verbünde Tupel die über Attribute mit übereinstimmenden Wertbelegungen zu selektieren besteht durch *äußere Verbünde* die Möglichkeit neben den Tupeln mit übereinstimmenden Werten alle Tupel einer am Verbund beteiligten Tabelle vollständig zu selektieren.

Definition 45: Äußerer Verbund

Ein Äußerer Verbund enthält die selektierten Daten aller beteiligten Tabellen, welche die formulierte Einschränkungsbefingung erfüllen, sowie alle Daten der „äußeren“ Tabelle. Die nicht mit Werten belegbaren Felder werden durch NULL aufgefüllt.

Konzeptionell wird zwischen *linken* und *rechten Äußeren Verbünden* unterschieden. Die „Seite“ des Verbundes gibt diejenige beteiligte Tabelle an, die im Rahmen der Verbundbildung vollständig ausgegeben wird.

[Beispiel 61](#) zeigt ein Beispiel eines linken Äußeren Verbundes, [Beispiel 62](#) illustriert einen rechten äußeren Verbund.

```
INSERT INTO EMPLOYEE VALUES("John", "X", "Doe", "999999999", "1965-03-04", "42 XYZ
Street", "M", 50000, NULL, NULL);
```

```
SELECT FNAME, DNAME
FROM EMPLOYEE LEFT OUTER JOIN DEPARTMENT
ON DEPARTMENT.DNUMBER = EMPLOYEE.DNO;
```

Beispiel 61: Linker Äußerer Verbund

Das Beispiel fügt zunächst einen Tupel zur Tabelle `EMPLOYEE` hinzu, der keiner Abteilung zugeordnet ist. In einem Inneren Verbund erscheint dieser Tupel daher nicht. Der linke Äußere Verbund des Beispiels hingegen umfaßt alle Tupel aus `EMPLOYEE` sowie die Werte der hinsichtlich der Bedingung `DEPARTMENT.DNUMBER = EMPLOYEE.DNO` ermittelten Übereinstimmungen in `DEPARTMENT`.

Für die nicht ermittelbaren Übereinstimmungen werden `NULL`-Werte erzeugt.

```
INSERT INTO DEPARTMENT VALUES("New Dept.", 0, 888665555, NULL);

SELECT FNAME, DNAME
FROM EMPLOYEE RIGHT OUTER JOIN DEPARTMENT
ON DEPARTMENT.DNUMBER = EMPLOYEE.DNO;
```

Beispiel 62: Rechter Äußerer Verbund

Auch das [Beispiel 62](#) zum rechten Äußeren Verbund fügt zunächst einen Datensatz ein; diesmal in die Tabelle `DEPARTMENT`, der zu keinem Tupel in `EMPLOYEE` in Beziehung steht. Analog dem linken Äußeren Verbund liefert der rechte Äußere Verbund alle Tupel der rechtsstehenden Tabelle (`DEPARTMENT`) sowie die mit `EMPLOYEE` übereinstimmenden.

Kreuzverbund

Der Kreuzverbund liefert alle gemäß den Gesetzen des kartesischen Produkts bildbaren Kombinationen aus Tupeln der beitragenden Relationen:

```
SELECT DNAME, PNUMBER
FROM DEPARTMENT CROSS JOIN PROJECT;
```

Beispiel 63: Kreuzverbund

Das Beispiel entspricht damit im Ergebnis der Anfrage aus [Beispiel 43](#).

Wird beim Kreuzverbund eine Bedingung angegeben, so entspricht er dem Inneren Verbund. Das Ergebnis der Anfrage aus [Beispiel 64](#) ist daher identisch zum inneren Verbund aus [Beispiel 56](#).

```
SELECT FNAME, DNAME
FROM DEPARTMENT CROSS JOIN EMPLOYEE
WHERE DEPARTMENT.DNUMBER = EMPLOYEE.DNO;
```

Beispiel 64: Kreuzverbund mit Bedingung

Artikel von Satya Komatineni: [The Effective Use of Joins in Select Statements](#)

Sortierungen

Zur Sortierung hinsichtlich einer oder mehrerer Spalten der als Anfrageergebnis ermittelten Tabelle steht die `ORDER BY`-Klausel zur Verfügung.

[Beispiel 65](#) zeigt die Anwendung zur lexikalischen Sortierung:

```
CREATE TABLE Person(
    Vorname VARCHAR(10),
    Nachname VARCHAR(10));

INSERT INTO Person VALUES("Adam", "C-Mann");
INSERT INTO Person VALUES("Cesar", "C-Mann");
INSERT INTO Person VALUES("Berta", "C-Mann");
INSERT INTO Person VALUES("Adam", "A-Mann");
INSERT INTO Person VALUES("Cesar", "A-Mann");
INSERT INTO Person VALUES("Berta", "A-Mann");
INSERT INTO Person VALUES("Adam", "B-Mann");
INSERT INTO Person VALUES("Cesar", "B-Mann");
INSERT INTO Person VALUES("Berta", "B-Mann");
```

```
SELECT *
FROM Person
ORDER BY Nachname;
```

Beispiel 65: Sortierung

Ist die Sortierung bezüglich mehrerer Attribute, d.h. Sortierung innerhalb eines gleicher Attributwerte hinsichtlich eines anderen Attributs, gewünscht, so können auch mehrere Sortierattribute in der ORDER BY-Klausel versammelt werden.

Zusätzlich zeigt das Beispiel die Kurzschreibweise, welche die zu sortierenden Attribute nicht namentlich benennt, sondern nur hinsichtlich ihrer Position innerhalb der SELECT-Klausel referenziert.

```
CREATE TABLE Person(
    Vorname VARCHAR(10),
    Nachname VARCHAR(10));

INSERT INTO Person VALUES("Adam", "C-Mann");
INSERT INTO Person VALUES("Cesar", "C-Mann");
INSERT INTO Person VALUES("Berta", "C-Mann");
INSERT INTO Person VALUES("Adam", "A-Mann");
INSERT INTO Person VALUES("Cesar", "A-Mann");
INSERT INTO Person VALUES("Berta", "A-Mann");
INSERT INTO Person VALUES("Adam", "B-Mann");
INSERT INTO Person VALUES("Cesar", "B-Mann");
INSERT INTO Person VALUES("Berta", "B-Mann");

SELECT *
FROM Person
ORDER BY 2,1;
```

Beispiel 66: Sortierung bezüglich mehrerer Attribute

Vorgabegemäß erfolgt die Sortierung aufsteigend (*ascending*). Die Umkehrung der Sortierreihenfolge kann durch nachstellen der Zeichenfolge DESC (für *descending*) nach dem Namen des Sortierattributs erreicht werden.

Die aufsteigende Vorgabesortierung (ASC) wird üblicherweise nicht ausgeschrieben, ist aber im [Beispiel 67](#) zur besseren Verdeutlichung expliziert.

```
CREATE TABLE Person(
    Vorname VARCHAR(10),
    Nachname VARCHAR(10));

INSERT INTO Person VALUES("Adam", "C-Mann");
INSERT INTO Person VALUES("Cesar", "C-Mann");
INSERT INTO Person VALUES("Berta", "C-Mann");
INSERT INTO Person VALUES("Adam", "A-Mann");
INSERT INTO Person VALUES("Cesar", "A-Mann");
INSERT INTO Person VALUES("Berta", "A-Mann");
INSERT INTO Person VALUES("Adam", "B-Mann");
INSERT INTO Person VALUES("Cesar", "B-Mann");
INSERT INTO Person VALUES("Berta", "B-Mann");

SELECT *
FROM Person
ORDER BY 2 ASC,1 DESC;
```

Beispiel 67: Auf- und Absteigende Sortierung bezüglich mehrerer Attribute**Unteranfragen**

Bisher wurden Anfragen lediglich auf Tabellen in ihrer Rolle als in der Datenbank abgelegte Eingabemengen betrachtet. Die relationale Sichtweise erfordert jedoch keineswegs, daß die Eingangswerte einer Anfrage direkt aus der Datenbank gelesen werden müssen. Sie können auch

Ergebnis einer weiteren Anfrage sein.

Anfragen die vor einer anderen Anfrage ausgeführt werden müssen um für diese Eingangswerte zu liefern werden daher als *Unterabfragen* (*subqueries* oder *nested queries*) bezeichnet.

Das [Beispiel 68](#) zeigt eine solche Unteranfrage die alle Projektnummern liefert welche Projekten zugeordnet sind die in der durch *Smith* geleiteten Abteilung bearbeitet werden. Eine zweite Unterabfrage des Beispiels liefert alle Nummern von Projekten an denen dieser Mitarbeiter selbst arbeitet.

Die durch diese Abfrage gelieferten Daten (Projektnummern) sind Eingangsdaten in die Ermittlung der Projektnamen.

```
SELECT DISTINCT PNAME
FROM PROJECT
WHERE PNUMBER IN (
  SELECT PNUMBER
  FROM PROJECT AS p, DEPARTMENT AS d, EMPLOYEE AS e
  WHERE e.SSN = d.MGRSSN AND
  d.DNUMBER = p.DNUM AND
  e.LNAME="Smith")
OR
  PNUMBER IN (SELECT PNO
  FROM WORKS_ON AS w, EMPLOYEE AS e
  WHERE w.ESSN = e.SSN AND
  e.LNAME="Smith");
```

Beispiel 68: Unterabfrage I

[Beispiel 69](#) zeigt den Vergleich eines Einzelwertes (SALARY) mit einer Menge gelieferter Werte. Die Anfrage ermittelt diejenigen Mitarbeiter, deren Einkommen höher liegt als das Einkommen aller Mitarbeit in Abteilung Nummer 5. (Hinweis es wird nicht ermittelt ob das Einkommen größer ist als die Summe aller Einkommen der Mitarbeiter aus Abteilung 5, sondern nur ob das Einkommen größer ist als jedes Einzeleinkommen eines Mitarbeiters aus Abteilung 5.)

```
SELECT LNAME, FNAME
FROM EMPLOYEE
WHERE SALARY > ALL (SELECT SALARY FROM EMPLOYEE WHERE DNO=5);
```

Beispiel 69: Unterabfrage II

Korrelierte Unteranfragen

Eine besondere Form der Unteranfragen stellen solche dar, die sich in ihrer WHERE-Klausel auf die äußere Anfrage beziehen.

Diese Form der Anfrageschachtelung wird auch als *korrelierte Unteranfrage* bezeichnet.

Das [Beispiel 70](#) zeigt eine solche Anfrage, die alle Verwandten (DEPENDENT) ermittelt, die das selbe Geschlecht haben wie der in der Tabelle EMPLOYEE erfaßte Mitarbeiter.

```
SELECT e.FNAME, e.LNAME
FROM EMPLOYEE AS e
WHERE e.SSN IN (SELECT ESSN
                FROM DEPENDENT
                WHERE e.SEX = SEX);
```

Beispiel 70: Korrelierte Unterabfrage

Jede korrelierte Unterabfrage kann durch Umschreibung in eine nicht-korrelierte Fassung überführt werden. So lautet die Formulierung des aus [Beispiel 70](#) ohne geschachtelte Unterabfrage:

```
SELECT e.FNAME, e.LNAME
FROM EMPLOYEE AS e, DEPENDENT AS d
WHERE e.SSN = d.ESSN AND
e.SEX = d.SEX;
```

Beispiel 71: Auflösung der korrelierten Unterabfrage

Die Formulierung als geschachtelte Unterabfrage ist damit nicht zwingend notwendig, kann jedoch aus Gründen der Übersichtlichkeit gewünscht sein.

Die nähere Betrachtung der Anfragen aus [Beispiel 70](#) und [Beispiel 71](#) zeigen, daß die aus der Tabelle DEPENDENT angefragten Daten lediglich zur Formulierung der Bedingung, nicht jedoch zur Ausgabe herangezogen werden. Daher läßt sich die Bedingung unter Verwendung des EXISTS-Operators umschreiben zu:

```
SELECT e.FNAME, e.LNAME
FROM EMPLOYEE AS e
WHERE EXISTS ( SELECT *
                FROM DEPENDENT
                WHERE e.SSN = ESSN AND
                e.SEX = SEX);
```

Beispiel 72: Korrelierte Unterabfrage mit EXISTS

EXISTS liefert den Boole'schen Wahrheitswert immer dann, wenn die (Unter-)Abfrage eine nichtleere Menge ist, d.h. Daten enthält.

Anfragen die EXISTS oder IN beinhalten können auch durch linke Äußere Verbünde ausgedrückt werden, wie [Beispiel 73](#) zeigt:

```
SELECT e.FNAME, e.LNAME
FROM EMPLOYEE AS e LEFT JOIN DEPENDENT AS d
ON e.SSN = d.ESSN AND e.SEX = d.SEX
WHERE d.SEX IS NOT NULL;
```

Beispiel 73: Korrelierte Unterabfrage ausgedrückt als linker äußerer Verbund

Eine ähnliche Funktion wie die EXISTS-Operation stellt ANY bereit, jedoch liefert diese die durch die Unterabfrage angefragten Tupel zurück um sie an eine Bedingung zu knüpfen.

[Beispiel 74](#) zeigt die Ermittlung der Namen derjenigen Mitarbeiter, die mehr als irgendein beliebiger Manager verdienen.

```
SELECT FNAME
FROM EMPLOYEE
WHERE SALARY > ANY (SELECT SALARY
                    FROM EMPLOYEE
                    WHERE SSN IN (SELECT SUPERSSN
                                FROM EMPLOYEE));
```

Beispiel 74: Unterabfrage unter Verwendung von ANY**Aggregatfunktionen und Gruppierung**

Über die Sortierung hinausgehend ist oftmals ein bestimmte Anordnung der durch eine Anfrage ermittelten Ergebnistupel gewünscht, etwa als inhaltliche Gruppierung.

Gleichzeitig sind oft quantitative Aussagen über Eigenschaften der Resultatmenge --- wie größter oder kleinster Wert sowie Summen- oder Durchschnittsbildung --- gewünscht.

[Beispiel 1](#) zeigt die Ermittlung der Summe aller Gehälter (SQL-Funktion SUM) sowie des Maximal- (MAX), Minimal- (MIN) und Durchschnittsgehalts (AVG) für die Mitarbeiter der *Research*-Abteilung. Die genannten SQL-Funktionen werden als *Aggregierungsfunktionen* bezeichnet, da sie die durch die Abfrage ermittelten Einzelwerte (d.h. die Einträge der Spalte SALARY) jeweils zu genau einer

Aussage verdichten.

```
SELECT SUM(SALARY), MAX(SALARY), MIN(SALARY), AVG(SALARY)
FROM EMPLOYEE, DEPARTMENT
WHERE DNO = DNUMBER AND DNAME="Research";
```

Beispiel 75: Aggregierungsfunktionen

Mit der Funktion COUNT steht eine Möglichkeit zur Ermittlung der Mächtigkeit einer Tupelmengende zur Verfügung. Beispiel [Beispiel 76](#) zeigt ihre Verwendung zur Ermittlung der Anzahl der Mitarbeiter der mit *Research* bezeichneten Abteilung.

```
SELECT COUNT(*)
FROM EMPLOYEE, DEPARTMENT
WHERE DNO = DNUMBER AND DNAME = "Research";
```

Beispiel 76: Zählfunktion I

Als Argument der COUNT-Funktion kann mit DISTINCT ein Schlüsselwort angegeben werden, welches die ausschließliche Zählung verschiedener Werte erwirkt. Die Anfrage aus Beispiel [Beispiel 77](#) ermittelt durch Nutzung dieses Schlüsselwortes die Anzahl der verschiedenen Werte in der Spalte SALARY.

```
SELECT COUNT(DISTINCT SALARY)
FROM EMPLOYEE;
```

Beispiel 77: Zählfunktion II

Häufig wird, wie in [Beispiel 78](#) gezeigt, eine Anfrage zur Ermittlung der Anzahl als Unterabfrage formuliert und in der umgebenden Hauptabfrage mit einer Bedingung versehen.

```
SELECT LNAME, FNAME
FROM EMPLOYEE
WHERE (SELECT COUNT(*)
      FROM DEPENDENT
      WHERE SSN=ESSN) >= 2;
```

Beispiel 78: Eingebettete Zählfunktion

Neben den bisher gezeigten aggregierten Aussagen über eine Gesamtmenge besteht oftmals der Wunsch nach von Ermittlung Aussagen dieses Stils über bestimmte Werteklassen innerhalb der betrachteten Gesamtmenge. Hierzu dienen Gruppierungen der Ausgangsmenge, auf welche dann die verschiedenen Aggregierungsfunktionen separat angewandt werden können. Beispiel [Beispiel 79](#) zeigt dies für die Ermittlung der Mitarbeiteranzahl pro Abteilung sowie der Berechnung des abteilungsinternen Durchschnittsgehalts.

```
SELECT d.DNAME AS "Abteilung", COUNT(*) AS "Anzahl Mitarbeiter", AVG(SALARY) AS
"Durchschnittsgehalt"
FROM EMPLOYEE AS e, DEPARTMENT AS d
WHERE e.DNO = d.DNUMBER
GROUP BY DNO;
```

Beispiel 79: Gruppierung

Zur Realisierung wird die GROUP BY-Klausel verwendet, welche die Angabe eines oder mehrerer Attribute zulässt anhand der die selektierte Menge partitioniert werden soll.

Die Anfrage des Beispiels [Beispiel 80](#) zeigt die Nutzung einer Verbundbedingung innerhalb einer Gruppierungsanfrage, die Projektnummer und -name sowie vermöge der COUNT-Funktion die Anzahl der das Projekt bearbeitenden Mitarbeiter ermittelt.

```
SELECT PNUMBER, PNAME, COUNT(*) AS "Anzahl Mitarbeiter"
FROM PROJECT, WORKS_ON
WHERE PNUMBER = PNO
GROUP BY PNUMBER, PNAME;
```

Beispiel 80: Gruppierung mit Verbundbedingung

Durch zusätzliche Angabe der HAVING-Klausel kann die Menge der Gruppierungsergebnisse mittels einer Bedingung beschränkt werden.

So ermittelt die Anfrage aus [Beispiel 81](#) dieselben Resultat wie die in [Beispiel 80](#) gezeigte, jedoch nur für Projekte deren Mitarbeiteranzahl größer 2 ist.

```
SELECT PNUMBER, PNAME, COUNT(*)
FROM PROJECT, WORKS_ON
WHERE PNUMBER = PNO
GROUP BY PNUMBER, PNAME
HAVING COUNT(*) > 2;
```

Beispiel 81: Bedingte Gruppierung

Die formulierte Beschränkung wirkt sich nicht auf die zur Berechnung herangezogene Grundgesamtheit, sondern lediglich auf die Ausgabe der Gruppierungsergebnisse aus, die vor der Auswertung der in der HAVING-Klausel formulierten Bedingung berechnet werden müssen.

Zur Beschränkung der zur Berechnung heranzuziehenden Grundgesamtheit steht auch unter Nutzung der GROUP BY-Klausel der durch WHERE formulierte Bedingungsteil der SELECT-Anfrage zur Verfügung.

```
SELECT PNUMBER, PNAME, COUNT(*)
FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE PNUMBER = PNO AND SSN = ESSN AND DNO=5
GROUP BY PNUMBER, PNAME;
```

Beispiel 82: Beschränkung der Gruppierungseingangsdaten

Gruppierungsschritte können auch in Unterabfragen auftreten, wie das [Beispiel 83](#) zur Ermittlung des Abteilungsnamens und der Anteil der darin arbeitenden Personen mit einem Gehalt über 40000 für alle Abteilungen mit mindestens 2 Mitgliedern zeigt:

```
SELECT DNAME, COUNT(*)
FROM DEPARTMENT, EMPLOYEE
WHERE DNUMBER = DNO AND SALARY > 4000 AND DNO IN (
    SELECT DNO
    FROM EMPLOYEE
    GROUP BY DNO
    HAVING COUNT(*) > 2)
GROUP BY DNUMBER;
```

Beispiel 83: Gruppierung in Unterabfrage

Der Datenmanipulationsteil von SQL

Neben den bisher betrachteten Eigenschaften der Sprache SQL zur Definition von Datenbankstrukturen und zur Abfrage von Datenbankinhalten stehen auch Befehle zur Manipulation in Form von Einfüge-, Aktualisierung- und Löschoperationen zur Verfügung.

Der Einfügebefehl INSERT

Zum Hinzufügen neuer Tupel in eine bestehende Tabelle durch Angabe von Werten für einen oder mehrere Spalten dieser Tabelle wird der Befehl INSERT angeboten.

Typischerweise wird dieser Befehlstyp zum Einfügen neuer Datensätze in bestehende Tabellen laufender Applikationen, ebenso wie zur Übernahme kompletter Datenbestände aus existierenden Datenquellen oder zur Neuladung einer Datenbank im Rahmen der Wiederherstellungsprozesses nach einem Systemausfall mit Datenverlust verwendet.

Die allgemeine Syntax des INSERT-Ausdruckes lautet:

```
INSERT INTO tbl_name (col_name,...)? VALUES(constant|NULL ...)
```

```
INSERT INTO EMPLOYEE VALUES(
    'John',
    'B',
    'Smith',
    123456789,
    '1965-01-09',
    '731 Fondren, Houston, TX',
    'M',
    30000,
    333445555,
    5);
```

Beispiel 84: Einfügen eines vollständigen Tupels

[Beispiel 84](#) zeigt den Befehl zur Erzeugung eines neuen Eintrages in der Tabelle EMPLOYEE der Demodatenbank. Die Aufzählung der einzufügenden Werte ist vollständig, d.h. für jede Spalte der Tabelle wird explizit ein konstanter Wert angegeben. Per Konvention müssen alle nichtnumerischen Werte in einfache oder doppelte Hochkommata eingeschlossen werden. Hierunter fallen neben den [Zeichenkettentypen](#) auch alle [Datumstypen](#).

Eine Sonderstellung innerhalb der angebbaren Konstanten zur Eintragung stellt die Zeichenkette NULL dar. Sie repräsentiert explizit fehlende Werte, deren Tabelleneinträge entsprechend gekennzeichnet werden. Zur Abgrenzung von der Zeichenkette NULL wird diese Angabe nicht in Anführungszeichen eingeschlossen, selbst wenn es sich um eine Spalte eines Zeichenkettentypen handelt.

[Beispiel 85](#) zeigt eine exemplarische Befehlskonstruktion:

```
INSERT INTO EMPLOYEE VALUES(
    'James',
    'E',
    'Borg',
    888665555,
    '1937-11-10',
    '450 Stone, Houston, TX',
    'M',
    55000,
    NULL,
    1);
```

Beispiel 85: Einfügen eines vollständigen Tupels mit NULL-Wert

Neben der Möglichkeit vollständige Tupel einzufügen, kann durch explizite Angabe der einzufügenden Spalten auch eine partielle Befüllung des neu erzeugten Tupels vorgenommen werden.

Für die im INSERT-Befehl nicht angegebenen Spalten wird der spezifizierte Vorgabewert oder NULL eingefügt.

[Beispiel 86](#) zeigt dies exemplarisch anhand des Einfügens der drei Attribute FNAME, LNAME und SSN. Gleichzeitig stellt das Beispiel auch heraus, daß bei expliziter Angabe der einzufügenden Spalten die gewählte Reihenfolge von der in der Tabelle realisierten abweichen kann.

```
INSERT INTO EMPLOYEE (LNAME, FNAME, SSN) VALUES(
    'Doe',
    'John',
    912873465);
```

Beispiel 86: Einfügen eines unvollständigen Tupels

Prinzipiell kann jedes Element der Potenzmenge der Attribute einer Relation eingefügt werden. Es muß jedoch zwingend einen Wert für das Primärschlüsselattribut enthalten, da hierfür der Wert NULL nicht gesetzt werden darf.

Der Aktualisierungsbefehl UPDATE

Zur Aktualisierung von Werten innerhalb bestehender Datenbankeinträge bietet der SQL-Sprachumfang den Befehl UPDATE an, der es gestattet frei wählbare Mengen von Tupeln einer Tabelle zu modifizieren.

Die allgemeine Syntax des Befehls lautet:

```
UPDATE tbl_name SET col_name=expression, ... [WHERE search_condition]
```

Beispiel [Beispiel 87](#) zeigt den Befehl zur null-Setzung aller in der Tabelle EMPLOYEE verwalteten Geburtsdaten (BDATE):

```
UPDATE EMPLOYEE SET BDATE=NULL;
```

Beispiel 87: Modifikation aller Tupel durch Setzen eines konstanten Wertes

Neben der Eintragung von Konstanten können auch neue Inhalte aus den Bisherigen errechnet werden. So zeigt [Beispiel 88](#) eine Aktualisierung, die das Gehalt (SALARY) aller Mitarbeiter um zehn Prozent erhöht:

```
UPDATE EMPLOYEE SET Salary=Salary*1.1;
```

Beispiel 88: Modifikation aller Tupel durch Setzen eines berechneten Wertes

Durch Nutzung der, identisch zum SELECT-Ausdruck aufgebauten, WHERE-Klausel kann die Menge der von der Änderung betroffenen Datensätze eingeschränkt werden.

Das Beispiel [Beispiel 89](#) ändert in allen Einträgen, deren LNAME auf Zelaya lautet den Wert zu Jones.

Die Anzahl der betroffenen Tupel ist durch den UPDATE-Ausdruck nicht festlegbar, sondern richtet sich ausschließlich nach der durch die WHERE-Klausel selektierten Eintragsmenge.

```
UPDATE EMPLOYEE SET LNAME='Jones'
WHERE LNAME='Zelaya';
```

Beispiel 89: Modifikation von Tupeln

Durch die Nutzbarkeit der vollständigen Möglichkeiten der aus dem SELECT-Befehl bekannten Mächtigkeit der WHERE-Klausel lassen sich selbst komplexe Aktualisierungen realisieren.

[Beispiel 90](#) zeigt führt die Erhöhung der Gehälter derjenigen Mitarbeiter durch, die Abteilungen zugewiesen sind, die mehr als zwei Projekte bearbeiten.

```
UPDATE EMPLOYEE SET SALARY=SALARY*1.1 WHERE DNO IN
(SELECT DNUMBER
FROM PROJECT AS p, DEPARTMENT AS d
WHERE d.DNUMBER=p.DNUM
GROUP BY 1
HAVING COUNT(*) > 2);
```

Beispiel 90: Modifikation von Tupeln (Ermittlung der betroffenen Tupel durch Subanfrage)

Der Löschbefehl DELETE

Zur Löschung von verwalteten Tupeln aus einer Tabelle existiert der DELETE-Befehl, der die betroffenen Datensätze ohne weite Nachfrage entfernt.

Seine allgemeine Syntax lautet:

```
DELETE FROM tbl_name [WHERE search_condition]
```

Die einfachste Ausprägung der DELETE-Anweisung löscht alle Tupel einer Tabelle:

```
DELETE FROM EMPLOYEE;
```

Beispiel 91: Löschen aller Tupel einer Tabelle

Durch Angabe der WHERE-Klausel können, wie bereits bei UPDATE für die zu aktualisierenden Tupel gezeigt, die zu löschenden Tupel eingegrenzt werden.

So entfernt der Ausdruck aus [Beispiel 92](#) alle Mitarbeiter die in Houston wohnen.

```
DELETE FROM EMPLOYEE
WHERE ADDRESS LIKE "%Houston%";
```

Beispiel 92: Löschen aller Mitarbeiter, die in Houston wohnhaft sind

Durch die Nutzung der expliziten Mengenangabe innerhalb der WHERE-Klausel läßt sich die Menge der zu entfernenden Datensätze statische eingrenzen wie [Beispiel 93](#) zeigt.

```
DELETE EMPLOYEE
WHERE SSN IN (333445555, 888665555, 987987987);
```

Beispiel 93: Löschen bestimmter Datenstätze

[Beispiel 94](#) zeigt die Nutzung einer Unterabfrage zur Ermittlung aller Abteilungen, die nur genau ein Projekt durchführen und anschließenden Löschung dieser Abteilungen aus der Tabelle DEPARTMENT.

```
DELETE FROM DEPARTMENT
WHERE DNUM IN
    (SELECT DNUM
     FROM PROJECT
     GROUP BY 1
     HAVING COUNT(*) = 1);
```

Beispiel 94: Löschen aller Abteilungen, die nur genau ein Projekt durchführen

▲ Definitionsverzeichnis

[Assoziation](#)

[Assoziationstyp](#)

[Äußerer Verbund](#)

[Boyce/Codd-Normalform](#)

[Daten](#)

[Datenbank](#)

[Datenbankmanagementsystem \(DBMS\)](#)

[Datenbanksprache](#)

[Datenunabhängigkeit](#)

[Dritte Normalform \(3NF\)](#)

[Entität](#)

[Entitätstyp](#)

[Erste Normalform \(1NF\)](#)

[Fünfte Normalform](#)

[Hybrider Entitäts-Assoziationstyp](#)

[Index](#)

[Information](#)

[Innerer Verbund](#)

[Kardinalitätsintervall](#)

[Konzeptuelles Schema](#)

[Logisches Schema](#)

[Mehrwertige Abhängigkeit](#)

[Metainformation](#)

[Modell](#)

[NULL-Wert](#)

[Physisches Schema](#)
[Primärschlüssel](#)
[Projektion](#)
[Referentielle Integrität](#)
[Relation](#)
[Relationales DBMS](#)
[Repräsentation](#)
[Repräsentationstyp](#)
[Rolle](#)
[Schlüssel](#)
[Selektion](#)
[Spezialisierungsassoziationstyp](#)
[Superschlüssel](#)
[Tabelle](#)
[Transitive Abhängigkeit](#)
[Triviale mehrwertige Abhängigkeit](#)
[Vierte Normalform](#)
[Volle funktionale Abhängigkeit](#)
[Vollständiges konzeptuelles Schema](#)
[Zweite Normalform \(2NF\)](#)

Schlagwortverzeichnis

[5NF](#)
[Aggregierungsfunktion](#)
[Aktualisierungsanomalie](#)
[Anomaliefreiheit](#)
[Anomalienfreiheit](#)
[Assoziation](#)
[Assoziationstyp](#)
[Atomarer Wert](#)
[Äußerer Verbund](#)
[BCNF](#)
[Boyce/Codd Normalform](#)
[Boyce/Codd-Normalform](#)
[data base](#)
[Data Control Language](#)
[Data Definition Language](#)
[Data Manipulation Language](#)
[Data Retrieval Language](#)
[Datenbank](#)
[Datenbankmanagementsystem \(DBMS\)](#)
[Datenbanksprache](#)
[Datenbankverwaltungssystem](#)
[Daten](#)
[Datenunabhängigkeit](#)
[DBMS](#)
[DBVS](#)
[DCL](#)
[DDL](#)
[Determinante](#)
[Diskursbereich](#)
[DKNF](#)
[DML](#)
[Domain-Key-Normalform](#)
[Domäne](#)
[Dritte Normalform \(3NF\)](#)
[DRL](#)
[Eindeutigkeitseinschränkung](#)
[Einfügeanomalie](#)
[Entität](#)
[Entitätstyp](#)
[Equi Joins](#)
[Erste Normalform \(1NF\)](#)

[Fünfte Normalform](#)
[fünfter Normalform](#)
[geschachtelter Relationen](#)
[Hybrider Entitäts-Assoziationstyp](#)
[inclusion dependence](#)
[Index](#)
[Information](#)
[Inklusionsabhängigkeit](#)
[Innerer Verbund](#)
[Kardinalitätsintervall](#)
[Konzeptuelles Schema](#)
[Logisches Schema](#)
[Löschanomalie](#)
[Mehrwertige Abhängigkeit](#)
[Metainformation](#)
[Metainformation](#)
[Miniwelt](#)
[Modell](#)
[multivalue dependency](#)
[MVD](#)
[NF2](#)
[NF2](#)
[NFNF](#)
[Non-First-Normal-Form](#)
[Normalformentheorie](#)
[NULL-Wert](#)
[Physisches Schema](#)
[PJNF](#)
[Primärschlüssel](#)
[Project Join Normalform](#)
[Projektion](#)
[RDBMS](#)
[Referentielle Integrität](#)
[Relationales DBMS](#)
[Relation](#)
[Repräsentation](#)
[Repräsentationstyp](#)
[Rolle](#)
[Schema](#)
[Schlüsselkandidat](#)
[Schlüssel](#)
[Selektion](#)
[Spezialisierungsassoziationstyp](#)
[spurious tupel](#)
[Superschlüssel](#)
[Tabelle](#)
[Template-Abhängigkeit](#)
[Transitive Abhängigkeit](#)
[triviale mehrwertige Abhängigkeit](#)
[Triviale mehrwertige Abhängigkeit](#)
[Universe of Discourse](#)
[Urrelation](#)
[Vierte Normalform](#)
[Volle funktionale Abhängigkeit](#)
[vollen funktionalen Abhängigkeit](#)
[Vollständiges konzeptuelles Schema](#)
[Zweite Normalform \(2NF\)](#)

Abbildungsverzeichnis

[3-Schema-Architektur](#)
[Entwicklungslinien des ER-Modells](#)
[Graphische Darstellung von Entitäten und Entitätstypen](#)
[Repräsentation und Repräsentationstyp](#)

[Identifizierende Repräsentationen](#)
[Assoziationen und Assoziationstypen](#)
[Vollständiges konzeptuelles Schema](#)
[Verschiedene Rollen](#)
[Hybrider Entitäts-Assoziationstyp](#)
[Informationsstruktur Adresse](#)
[Spezialisierung](#)
[Metainformation](#)
[Metainformation](#)
[Konzeptuelles Schema der Fallstudie](#)
[Über Fremdschlüssel verknüpfte Relationen](#)
[Voll funktionale Abhängigkeiten in der dargestellten Relation](#)
[Relationen in 2NF](#)
[Konzeptuelles Schema in E3R-Notation für die betrachteten Zusammenhänge](#)
[Transitive Abhängigkeiten](#)
[Relation in 3NF](#)
[Konzeptuelles Schema in E3R-Notation für die betrachteten Zusammenhänge](#)
[Funktionale Abhängigkeiten](#)
[Relation in BCNF](#)
[Konzeptuelles Schema in E3R-Notation für die betrachteten Zusammenhänge](#)
[Mehrwertige Abhängigkeiten](#)
[Relation in 4NF](#)
[Konzeptuelles Schema in E3R-Notation für die betrachteten Zusammenhänge](#)
[Konzeptuelles Schema in E3R-Notation für die betrachteten Zusammenhänge](#)

Verzeichnis der Beispiele

[Am Markt verfügbare DBM-Systeme](#)
[Am Markt verfügbare RDBM-Systeme](#)
[Relationen](#)
[Tabelle](#)
[Modelle](#)
[Die Datenbanksprache SQL](#)
[Relationen sind ein logisches Schema](#)
[Beispiele für Entitäten](#)
[Beispiele für Entitätstypen](#)
[Beispiele für Repräsentationstypen](#)
[Beispiele für Repräsentationen](#)
[Beispiele für Assoziationstypen](#)
[Beispiele für Kardinalitätsintervalle](#)
[Beispiele für Superschlüssel](#)
[Beispiele für Schlüssel](#)
[Beispiele für Superschlüssel](#)
[Beispiel für referentielle Integrität](#)
[Geschwindigkeitsverhalten mit/ohne Index](#)
[Relation, die nicht in 1NF ist](#)
[Relation, die in 1NF ist](#)
[Erzeugung einer Tabelle](#)
[Ermittlung von Tabelleninformation](#)
[Erzeugung einer temporären Tabelle](#)
[Auswirkung von Datentypen I](#)
[Auswirkung von Datentypen II](#)
[Auswirkung von Datentypen III](#)
[Auswirkung von Datentypen IV](#)
[Auswirkung von Datentypen V](#)
[Auswirkung von Datentypen VI](#)
[Definition einer Spalte als NOT NULL](#)
[Definition einer Spalte als NULL](#)
[Definition einer Spalte mit Vorgabewerten](#)
[Definition eines Primärschlüssels](#)
[Definition eines zusammengesetzten Primärschlüssels](#)
[Definition eines automatisch befüllten Primärschlüssels](#)
[Definition von Indexen](#)
[Erzeugung von Fremdschlüsselbeziehungen zum Tabellenerstellungszeitpunkt](#)

[Nachträgliche Erzeugung von Fremdschlüsselbeziehungen](#)

[Einfache Anfrage](#)

[Anfrage aller Spalten einer Tabelle](#)

[Anfrage aller Spalten einer Tabelle mit Jokerzeichen](#)

[Duplikatfreie Ausgabe aller verschiedenen Werte](#)

[Anfrage auf zwei Tabellen](#)

[Fehlerhafte Anfrage auf zwei Tabellen](#)

[Lösung des Mehrdeutigkeitsproblems bei Anfrage auf zwei Tabellen](#)

[Lösung des Mehrdeutigkeitsproblems bei Anfrage auf zwei Tabellen](#)

[Umbenennung von Ausgabespalten](#)

[Berechnungen I](#)

[Einschränkung der Anfrage](#)

[Musterbasierte Anfrage I](#)

[Musterbasierte Anfrage II](#)

[Musterbasierte Anfrage III](#)

[Kombination von Bedingungen](#)

[Kombination mittels UNION](#)

[Fehlerhafte Verbundbildung](#)

[Innerer Verbund](#)

[Innerer Verbund in Standardnotation](#)

[Innerer Verbund unter mehrfacher Nutzung derselben Tabelle](#)

[Innerer Verbund dreier Tabellen](#)

[Non-Equi-Join](#)

[Linker Äußerer Verbund](#)

[Rechter Äußerer Verbund](#)

[Kreuzverbund](#)

[Kreuzverbund mit Bedingung](#)

[Sortierung](#)

[Sortierung bezüglich mehrerer Attribute](#)

[Auf- und Absteigende Sortierung bezüglich mehrerer Attribute](#)

[Unterabfrage I](#)

[Unterabfrage II](#)

[Korrelierte Unterabfrage](#)

[Auflösung der korrelierten Unterabfrage](#)

[Korrelierte Unterabfrage mit EXISTS](#)

[Korrelierte Unterabfrage ausgedrückt als linker äußerer Verbund](#)

[Unterabfrage unter Verwendung von ANY](#)

[Aggregierungsfunktionen](#)

[Zählfunktion I](#)

[Zählfunktion II](#)

[Eingebettete Zählfunktion](#)

[Gruppierung](#)

[Gruppierung mit Verbundbedingung](#)

[Bedingte Gruppierung](#)

[Beschränkung der Gruppierungseingangsdaten](#)

[Gruppierung in Unterabfrage](#)

[Einfügen eines vollständigen Tupels](#)

[Einfügen eines vollständigen Tupels mit NULL-Wert](#)

[Einfügen eines unvollständigen Tupels](#)

[Modifikation aller Tupel durch Setzen eines konstanten Wertes](#)

[Modifikation aller Tupel durch Setzen eines berechneten Wertes](#)

[Modifikation von Tupeln](#)

[Modifikation von Tupeln \(Ermittlung der betroffenen Tupel durch Subanfrage\)](#)

[Löschen aller Tupel einer Tabelle](#)

[Löschen aller Mitarbeiter, die in Houston wohnhaft sind](#)

[Löschen bestimmter Datenstände](#)

[Löschen aller Abteilungen, die nur genau ein Projekt durchführen](#)

Service provided by [Mario Jeckle](#)

Generated: 2004-01-08T22:57:50+01:00

 [Feedback](#)  [SiteMap](#)

 [This page's original location: http://www.jeckle.de/vorlesung/datenbanken/script.html](#)

 [RDF description for this page](#)

```
import java.sql.Array;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class JDBCSelect8 {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException cnfe) {
            System.err.println("Driver class not found");
            cnfe.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException sqle) {
            System.err.println("Error establishing database connection");
            sqle.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt = (Statement) con.createStatement();
        } catch (SQLException sqle) {
            System.err.println("Error creating SQL-Statement");
            sqle.printStackTrace();
        }

        try {
            ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
            while (!rs.isLast()) {
                rs.first();
                System.out.print(rs.getString("FNAME") + "\t");
                Array cars = rs.getArray("CAR");
                ResultSet carsRS = cars.getResultSet();
                System.out.print("(");
                while (!carsRS.isLast()) {
```

```
        rs.first();
        System.out.print(carsRS.getString("CAR"));
        carsRS.next();
    }
    System.out.println("");
    rs.next();
}
} catch (SQLException sqle) {
    System.err.println("Error selecting values from table EMPLOYEE");
    sqle.printStackTrace();
}
}
```

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class JDBCSelect9 {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException cnfe) {
            System.err.println("Driver class not found");
            cnfe.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException sqle) {
            System.err.println("Error establishing database connection");
            sqle.printStackTrace();
        }

        try {
            File file = new File(args[0]);
            FileInputStream fis = new FileInputStream(args[0]);
            PreparedStatement pstmt =
                con.prepareStatement(
                    "UPDATE EMPLOYEE SET binData =? WHERE SSN=123456789");
            pstmt.setBinaryStream(1, fis, (int) file.length());
            pstmt.executeUpdate();
            fis.close();

            //read it back from the database
            Statement stmt = (Statement) con.createStatement();
```

```
ResultSet rs =  
    stmt.executeQuery(  
        "SELECT binData FROM EMPLOYEE WHERE SSN='123456789'");
```

```
FileOutputStream fos = new FileOutputStream(args[1]);  
if (rs.next())  
    fos.write(rs.getBytes(1));  
fos.close();
```

```
} catch (SQLException sqle) {  
    System.err.println("Error selecting values from table EMPLOYEE");  
    sqle.printStackTrace();  
} catch (IOException ioe) {  
    ioe.printStackTrace();  
}
```

```
}
```

```
}
```

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;

import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class JDBCTransact1 {
    private static void printContent(Statement stmt) throws SQLException {
        ResultSet rs =
            stmt.executeQuery("SELECT FNAME,MINIT,LNAME FROM EMPLOYEE;");
        while (!rs.isLast()) {
            rs.next();
            System.out.println(
                rs.getString("LNAME")
                + "\t"
                + rs.getString("MINIT")
                + "\t"
                + rs.getString("LNAME"));
        }
    }

    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException cnfe) {
            System.err.println("Driver class not found");
            cnfe.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException sqle) {
            System.err.println("Error establishing database connection");
            sqle.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt = (Statement) con.createStatement();
        } catch (SQLException sqle) {
```

```
        System.err.println("Error creating SQL-Statement");
        sqle.printStackTrace();
    }
```

```
try {
    int transactionIsolation = con.getTransactionIsolation();
    switch (transactionIsolation) {
        case Connection.TRANSACTION_NONE :
            System.out.println("Transactions are not supported");
            break;
        case Connection.TRANSACTION_READ_UNCOMMITTED :
            System.out.println(
                "Dirty reads, non-repeatable reads and phantom reads can occur");
            break;
        case Connection.TRANSACTION_READ_COMMITTED :
            System.out.println(
                "Dirty reads are prevented; non-repeatable reads and phantom reads can occur");
            break;
        case Connection.TRANSACTION_REPEATABLE_READ :
            System.out.println(
                "Dirty reads and non-repeatable reads are prevented; phantom reads can occur");
            break;
        case Connection.TRANSACTION_SERIALIZABLE :
            System.out.println(
                "Dirty reads, non-repeatable reads and phantom reads are prevented");
            break;
    }
    if (transactionIsolation < Connection.TRANSACTION_SERIALIZABLE) {
        con.setTransactionIsolation(
            Connection.TRANSACTION_SERIALIZABLE);
        if (con.getTransactionIsolation()
            != Connection.TRANSACTION_SERIALIZABLE) {
            System.out.println(
                "cannot set Connection.TRANSACTION_SERIALIZABLE");
        } else {
            System.out.println(
                "reached highest possible isolation level");
        }
    }
}
```

```
con.setAutoCommit(false);
stmt.executeUpdate(
    "INSERT INTO EMPLOYEE VALUES('Hans','X','Hinterhuber','11111111',
    NULL,NULL,NULL,NULL,NULL,NULL);");
```

```
        stmt.executeUpdate(
            "INSERT INTO EMPLOYEE VALUES('Franz','X','Obermüller','22222222',
NULL,NULL,NULL,NULL,NULL,NULL);");
        printContent(stmt);
        //suppose error happens here
        Thread.sleep(5000);
        boolean error = true;
        if (error) {
            con.rollback();
        } else {
            stmt.executeUpdate(
                "INSERT INTO EMPLOYEE VALUES('Fritz','X','Meier','33333333',
NULL,NULL,NULL,NULL,NULL,NULL);");
        }
        printContent(stmt);
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    } catch (InterruptedException ie) {
        ie.printStackTrace();
    }
}
```


Dirty reads and non-repeatable reads are prevented; phantom reads can occur
reached highest possible isolation level

Smith B	Smith
Wong T	Wong
English A	English
Narayan K	Narayan
Borg E	Borg
Wallace S	Wallace
Jabbar V	Jabbar
Zelaya J	Zelaya
Smith B	Smith
Wong T	Wong
English A	English
Narayan K	Narayan
Borg E	Borg
Wallace S	Wallace
Jabbar V	Jabbar
Zelaya J	Zelaya

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Savepoint;

import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class JDBCTransact2 {
    private static void printContent(Statement stmt) throws SQLException {
        ResultSet rs =
            stmt.executeQuery("SELECT FNAME,MINIT,LNAME FROM EMPLOYEE;");
        while (!rs.isLast()) {
            rs.next();
            System.out.println(
                rs.getString("LNAME")
                + "\t"
                + rs.getString("MINIT")
                + "\t"
                + rs.getString("LNAME"));
        }
    }

    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException cnfe) {
            System.err.println("Driver class not found");
            cnfe.printStackTrace();
        }

        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://localhost/jdbctest/",
                    "mario",
                    "thePassword");
        } catch (SQLException sqle) {
            System.err.println("Error establishing database connection");
            sqle.printStackTrace();
        }

        Statement stmt = null;
        try {
            stmt = (Statement) con.createStatement();
        }
```

```
} catch (SQLException sqle) {
    System.err.println("Error creating SQL-Statement");
    sqle.printStackTrace();
}

try {
    int transactionIsolation = con.getTransactionIsolation();
    switch (transactionIsolation) {
        case Connection.TRANSACTION_NONE :
            System.out.println("Transactions are not supported");
            break;
        case Connection.TRANSACTION_READ_UNCOMMITTED :
            System.out.println(
                "Dirty reads, non-repeatable reads and phantom reads can occur");
            break;
        case Connection.TRANSACTION_READ_COMMITTED :
            System.out.println(
                "Dirty reads are prevented; non-repeatable reads and phantom reads can occur");
            break;
        case Connection.TRANSACTION_REPEATABLE_READ :
            System.out.println(
                "Dirty reads and non-repeatable reads are prevented; phantom reads can occur");
            break;
        case Connection.TRANSACTION_SERIALIZABLE :
            System.out.println(
                "Dirty reads, non-repeatable reads and phantom reads are prevented");
            break;
    }
    if (transactionIsolation < Connection.TRANSACTION_SERIALIZABLE) {
        con.setTransactionIsolation(
            Connection.TRANSACTION_SERIALIZABLE);
        if (con.getTransactionIsolation()
            != Connection.TRANSACTION_SERIALIZABLE) {
            System.out.println(
                "cannot set Connection.TRANSACTION_SERIALIZABLE");
        } else {
            System.out.println(
                "reached highest possible isolation level");
        }
    }
}

con.setAutoCommit(false);
stmt.executeUpdate(
    "INSERT INTO EMPLOYEE VALUES('Hans','X','Hinterhuber','111111111',
```

```
NULL,NULL,NULL,NULL,NULL,NULL));");
    Savepoint sp = con.setSavepoint();
    stmt.executeUpdate(
        "INSERT INTO EMPLOYEE VALUES('Franz','X','Obermüller','22222222',
NULL,NULL,NULL,NULL,NULL,NULL));");
    printContent(stmt);
    //suppose error happens here
    Thread.sleep(5000);
    boolean error = true;
    if (error) {
        con.rollback(sp);
    }
    stmt.executeUpdate(
        "INSERT INTO EMPLOYEE VALUES('Fritz','X','Meier','33333333',NULL,
NULL,NULL,NULL,NULL,NULL));");
    printContent(stmt);
    con.commit();
} catch (SQLException sqle) {
    sqle.printStackTrace();
} catch (InterruptedException ie) {
    ie.printStackTrace();
}
}
```

searchSecurity.com Definitions - powered by whatis.com

BROWSE [WHATIS.COM](#) **DEFINITIONS:** [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#) <#>

[BROWSE ALL CATEGORIES](#)

Search [whatis.com](#) for: - OR - Search this site:

Secure Shell

powered by  [whatismyip.com](#)

The term you selected is being presented by searchSecurity.com, a TechTarget site for Security professionals.

SITE SPONSOR

Secure Shell (SSH), sometimes known as Secure Socket Shell, is a [Unix](#)-based command interface and [protocol](#) for securely getting access to a remote computer. It is widely used by network administrators to control Web and other kinds of servers remotely. SSH is actually a suite of three utilities - slogin, ssh, and scp - that are secure versions of the earlier UNIX utilities, [rlogin](#), rsh, and rcp. SSH commands are encrypted and secure in several ways. Both ends of the [client/server](#) connection are authenticated using a [digital certificate](#), and passwords are protected by being encrypted.

SSH uses [RSA public key](#) cryptography for both connection and authentication. Encryption algorithms include [Blowfish](#), [DES](#), and [IDEA](#). IDEA is the default.

SSH2, the latest version, is a proposed set of standards from the Internet Engineering Task Force ([IETF](#)).

Read more about it:

- >> [Steve Acheson's Secure Shell \(SSH\) Frequently Asked Questions](#) is recommended.
- >> [Rajpaul Bagga offers a Secure Shell \(SSH\) Howto](#).
- >> [The developer of SSH is SSH Communications Security](#).

Last updated on: Apr 22, 2003

<< [Back to previous page](#) [Go to whatis.com home page](#) >>

EXPLORE THIS AREA: RICH-MEDIA ADVERTISEMENT

WHAT'S NEW

on searchSecurity

1. [Access career resources](#)
2. [Find a firewall solution here](#)
3. [Win a Mercedes!](#)

SECURITY MARKETPLACE

 [GET YOUR PRODUCT OR SERVICE LISTED HERE](#)

[Managed Email Security Services to Businesses Worldwide: MessageLabs](#)

MessageLabs email security system operates at the Internet level, to keep threats away from customers networks, and serves as a first line of defense against spam, viruses and other unwanted content. The companies anti-virus services provides unparalleled, 100% protection against known and unknown viruses. Delivered on a global infrastructure, the services are easy to provision, highly available, fully redundant and supported 24x7.

[Internet Access Reporting](#)

Monitor employee Internet use accurately with Cyfin software. Prevent workplace Web abuse, track bandwidth usage and quickly generate clear, customized reports on Internet activity.

[Enterprise Content Security from FutureSoft](#)

Internet filtering, e-mail filtering and file surveillance with the DynaComm i:series product family from FutureSoft provides your organization with a complete enterprise content security solution. Learn more and download free trials today.

[ThreatSentry: Neural IPS protects IIS web servers.](#)

Breakthrough neural application compares system requests against an evolving baseline to prevent known, undocumented and other misuse for Microsoft? IIS. Free 30-day trial and install/eval session.

[Need a MS Exchange Spam Filter??](#)

Spam sucks. Your life shouldn't. iHateSpam SE was uniquely developed to be both user- and admin-friendly. Control spam according to the

needs of your company and users. V5.5, 2000,Exchange 2003, and Gateway. Reader's Choice Winner 2003, Best Antispam Tool.
[The Quick Guide to Windows Server 2003. Get the latest information on installation, new features and training. Click here!](#)
[Try a free session of the new Microsoft Visual Studio .NET 2003--no download required. http://msdn.microsoft.com/vstudio](#)

 [GET YOUR PRODUCT OR SERVICE LISTED HERE](#)



The Source for Java Developers



Developers Home > Products & Technologies > Java Technology > JDBC

Profile and Registration | Why Register?

J2EE JDBC Technology

Downloads

- Early Access
- Tools

Reference

- API Specifications
- Documentation
- FAQs
- Code Samples and Apps
- Technical Articles and Tips
- Industry Support

Community

- Bookshelf
- Code Certification
- Bug Database
- Forums

Learning

- Tutorial and Code Camps
- Online Courses
- Instructor-Led Courses
- Quizzes

Regional Sites - 日本語

JDBC technology is an API (included in both J2SE and J2EE releases) that provides cross-DBMS connectivity to a wide range of SQL databases and access to other tabular data sources, such as spreadsheets or flat files. With a JDBC technology-enabled driver, you can connect all corporate data even in a heterogeneous environment. » [Read More](#)

Notes for Developers

Interested in certifying your JDBC driver or finding a JDBC driver for use with J2EE compatible products? Check out the [JDBC certification program](#). Our [driver database](#) is now searchable by supported JDBC API version, driver type, DBMS support, and features.

See also our page covering the relationship of JDBC to a related technology: [Java Data Objects \(JDO\)](#).

What's New

12 December 2003

JDBC RowSet Implementations - Proposed Final Draft

Available through the Java Community Process (JCP) site, this spec defines the APIs for implementations of the Rowset interface for passing tabular data between distributed tiers and components. Also defines the XML Schema that will be used to represent the Rowset objects that are passed between components.

» [Read more](#)

10 June 2003

JDBC 4.0 Specification - JSR 221 Filed The JDBC 4.0 API specification seeks to improve Java application access to SQL data stores by the provision of ease-of-development focused features and improvements at both the utility and API level. Using new Java language features planned for JSR-175, generics defined in JSR-014, a set of JDBC utility classes, SQL-savvy developers will be able to more easily access SQL data sources while still benefiting from the full power of the JDBC API.

» [Read more](#)

11 November 2003

JDBC Optional Package for CDC/Foundation Profile - Final Approval Ballot

The JDBC API for CDC / FP Optional Package defines a subset of the JDBC 3.0 API that can be used in conjunction with the Java 2 Micro Edition (J2ME) Connected Device Configuration / Foundation Profile (CDC / FP).

9 May 2002

JDBC API 3.0 Specification - Final Release The JDBC API v. 3.0 is included in J2SE 1.4. The JDBC Optional Package will be packaged with the core API and be included as part of J2SE. Other major changes in the new version of the JDBC

Related Links

Popular Downloads

- J2SE 1.4.2
- Java Web Services Developer Pack 1.3

Technical Topics

- Performance
- Web Services
- Security
- Desktop

Products and Technologies

- J2SE 1.5 (coming soon)
- J2SE 1.4.2
- Desktop Java
- Java Web Services
- J2EE
- Java Dynamic Management Kit (JDMK)
- Sun Java Studio Standard IDE

Sun Resources

- [java.net](#)
- [Java Upgrade Program](#)
- [New to Java Center](#)
- [Professional Certification](#)
- [Professional Training](#)
- [JavaOne Online](#)

specification include: Connection Pool configuration enhancements; Statement pooling for Pooled Connections; and a description of the migration path from the JDBC SPI (Service Provider Interface) to the Connector Architecture.

24 October 2002

JDBC Connector Early Access The JDBC Connector enables any JDBC driver to be packaged as a J2EE Connector Architecture-compliant connector, and it makes it easy to plug any JDBC driver into an application server compliant with the J2EE platform.

Community

Events

2004 JavaOne Conference. San Francisco , CA Be there! Join the thousands of developers worldwide who come to the JavaOne conference each year in San Francisco to immerse themselves in Java technology, the latest innovations, the community, and the learning opportunities. » [Read More](#)

Subscribe to Newsletters. Members of [Sun Developer Network](#) can [sign up](#) to receive these (and other) newsletters. Not yet a member? [Join us!](#)

Enterprise Java Technologies Newsletter

Learn about new enterprise Java technologies, products, tools, and resources for developers.

Enterprise Java Technologies Tech Tips

Get expert tips, sample code solutions, and techniques for developing in the Java 2 Platform, Enterprise Edition (J2EE).

» [Read More](#)

JDBC Interest

JDBC API Mailing List Learn more about the JDBC API. To subscribe, send email to listserv@java.sun.com with this info in the body of the message: subscribe JDBC-INTEREST yourlastname yourfirstname.



Carol McDonald, Staff Engineer, Code Camp Engineer in J2EE, Web Services and Integration



[Company Info](#) | [About SDN](#) | [Press](#) | [Contact Us](#) | [Employment](#)
[How to Buy](#) | [Licensing](#) | [Terms of Use](#) | [Privacy](#) | [Trademarks](#)

Copyright 1994-2004 Sun Microsystems, Inc.

A Sun Developer Network Site

Unless otherwise licensed, code in all technical manuals herein (including articles, FAQs, samples) is provided under this [License](#).

[XML](#) [Content Feeds](#)



The Source for Java Developers



[Developers Home](#) > [Products & Technologies](#) > [Java Technology](#) > [J2EE](#) > [JDBC](#) > [Learning](#) > [Tutorial and Code Camps](#) >

[Profile and Registration](#) | [Why Register?](#)

Tutorials & Code Camps

Aids for Learning to Use the JDBC API

[Printable Page](#)

Educational Materials

Aids for Learning to Use the JDBC API

- Java Series book *[JDBC API Tutorial and Reference, Third Edition](#)* covers all of the JDBC 3.0 API, which includes the core API and the Optional Package API. It offers tutorials on basic features, advanced features, using metadata, and using rowsets. The reference section explains much more than is found in the specification and is the definitive reference for the JDBC API.
- Online Tutorials -- based on the tutorials in the Java Series book
 - [Basic Tutorial](#)
 - [Advanced Tutorial](#)
 - [Rowset Tutorial](#)
- [Online Tutorial](#) -- from the Java Developer ConnectionSM



[Company Info](#) | [About SDN](#) | [Press](#) | [Contact Us](#) | [Employment](#)
[How to Buy](#) | [Licensing](#) | [Terms of Use](#) | [Privacy](#) | [Trademarks](#)

Copyright 1994-2004 Sun Microsystems, Inc.

A Sun Developer Network Site

Unless otherwise licensed, code in all technical manuals herein (including articles, FAQs, samples) is provided under this [License](#).

[Content Feeds](#)

**Trail:** JDBC(TM) Database Access**Lesson:** New Features in the JDBC 2.0 API

Inserting and Deleting Rows Programmatically

In the previous section you saw how to modify a column value using methods in the JDBC 2.0 API rather than having to use SQL commands. With the JDBC 2.0 API, you can also insert a new row into a table or delete an existing row programmatically.

Let's suppose that our coffee house proprietor is getting a new variety from one of his coffee suppliers, The High Ground, and wants to add the new coffee to his database. Using the JDBC 1.0 API, he would write code that passes an SQL insert statement to the DBMS. The following code fragment, in which `stmt` is a `Statement` object, shows this approach:

```
stmt.executeUpdate("INSERT INTO COFFEES " +  
                  "VALUES ( 'Kona', 150, 10.99,  
0, 0)");
```

You can do the same thing without using any SQL commands by using `ResultSet` methods in the JDBC 2.0 API. Basically, after you have a `ResultSet` object with results from the table `COFFEES`, you can build the new row and then insert it into both the result set and the table `COFFEES` in one step. You build a new row in what is called the insert row, a special row associated with every `ResultSet` object. This row is not actually part of the result set; you can think of it as a separate buffer in which to compose a new row.

Your first step will be to move the cursor to the insert row, which you do by invoking the method `moveToInsertRow`. The next step is to set a value for each column in the row. You do this by calling the appropriate `updateXXX` method for each value. Note that these are the same `updateXXX` methods you used in the previous section for changing a column value. Finally, you call the method `insertRow` to insert the row you have just populated with values into the result set. This one method simultaneously inserts the row into both the `ResultSet` object and the database table from which the result set was selected.

The following code fragment creates the scrollable and updatable `ResultSet` object `uprs`, which contains all of the rows and columns in the table `COFFEES`:

```

Connection con = DriverManager.getConnection
("jdbc:mySubprotocol:mySubName");
Statement stmt = con.createStatement(ResultSet.
TYPE_SCROLL_SENSITIVE,
                                ResultSet.
CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("SELECT *
FROM COFFEES");

```

The next code fragment uses the `ResultSet` object `uprs` to insert the row for Kona coffee, shown in the SQL code example. It moves the cursor to the insert row, sets the five column values, and inserts the new row into `uprs` and `COFFEES` :

```

uprs.moveToInsertRow();
uprs.updateString("COF_NAME", "Kona");
uprs.updateInt("SUP_ID", 150);
uprs.updateFloat("PRICE", 10.99);
uprs.updateInt("SALES", 0);
uprs.updateInt("TOTAL", 0);
uprs.insertRow();

```

Because you can use either the column name or the column number to indicate the column to be set, your code for setting the column values could also have looked like this:

```

uprs.updateString(1, "Kona");
uprs.updateInt(2, 150);
uprs.updateFloat(3, 10.99);
uprs.updateInt(4, 0);
uprs.updateInt(5, 0);

```

You might be wondering why the `updateXXX` methods seem to behave differently here from the way they behaved in the update examples. In those examples, the value set with an `updateXXX` method immediately replaced the column value in the result set. That was true because the cursor was on a row in the result set. When the cursor is on the insert row, the value set with an `updateXXX` method is likewise immediately set, but it is set in the insert row rather than in the result set itself. In both updates and insertions, calling an `updateXXX` method does not affect the underlying database table. The method `updateRow` must be called to have updates occur in the database. For insertions, the method `insertRow` inserts the new row into the result set and the database at the same time.

You might also wonder what happens if you insert a row but do not supply a value for every column in the row. If you fail to supply a value for a column that was defined to accept SQL `NULL` values, then the value assigned to that column is `NULL`. If a column does not accept null values, however, you will get an `SQLException` when you do not call an `updateXXX` method to set a value for it. This is also true if a table column is missing in your `ResultSet` object. In the example above, the query was `SELECT * FROM COFFEES`, which produced a result set with all the columns of all the rows. When you want to insert one or more rows, your query does not have to select all rows, but it is safer to select all columns. Especially if your table has hundreds or thousands of rows, you might want to use a `WHERE` clause to limit the number of rows returned by your `SELECT` statement.

After you have called the method `insertRow`, you can start building another row to be inserted, or you can move the cursor back to a result set row. You can, for instance, invoke any of the methods that put the cursor on a specific row, such as `first`, `last`, `beforeFirst`, `afterLast`, and `absolute`. You can also use the methods `previous`, `relative`, and `moveToCurrentRow`. Note that you can invoke `moveToCurrentRow` only when the cursor is on the insert row.

When you call the method `moveToInsertRow`, the result set records which row the cursor is sitting on, which is by definition the current row. As a consequence, the method `moveToCurrentRow` can move the cursor from the insert row back to the row that was previously the current row. This also explains why you can use the methods `previous` and `relative`, which require movement relative to the current row.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

<a href="http://java.sun.com/docs/books/tutorial/jdbc/jdbc2dot0/inserting.html" (3 of 3)09.01.2004 07:41:40


[REGISTER](#)

 EMAIL

 PASSWORD
[GO](#)
[Forgot your password?](#)
[HOME](#)

[FAQS](#)
[FORUMS](#)
[DOWNLOADS](#)
[ARTICLES](#)
[PEERSCOPE](#)
[LEARN](#)
[JAVAPRO](#)

 Search
[GO](#) [?](#)

JDBC FAQ Home Page

 View:
[GO](#)
[ASK A QUESTION](#)

FAQ Manager is guru [Joe Sam Shirah](#) [PREMIUM](#).

Java Database Connectivity is the standard for communication between a Java application and a relational database. The JDBC API is released in two versions; JDBC version 1.22 (released with JDK 1.1.X in package java.sql) and version 2.0 (released with Java platform 2 in packages java.sql and javax.sql). It is a simple and powerful largely database-independent way of extracting and inserting data to or from any database. [FAQ Previously managed by Lennart Jorelid.]

What's New

- **What is the best way to generate a universally unique object ID? Do I need to use an external resource like a file or database, or can I do it all in memory?**
Languages:Markup:XML, Java:API:Servlets, Java:API:JDBC, Java:API:EJB:
EntityBean:Primary Keys, Process:Patterns:BestPractices
Alessandro A. Garbagnati [PREMIUM](#), Nov 25, 2002
 [I need to generate unique id's that will be used for node 'ID' attribute values within XML documents. This id must be unique system-wide. The generator...
- **Whan happens when I close a Connection application obtained from a connection Pool? How does a connection pool maintain the Connections that I had closed through the application?**
Java:API:JDBC:Connections
Christopher Koenigsberg [PREMIUM](#), Apr 14, 2002
 It is the magic of polymorphism, and of Java interface vs. implementation types. Two objects can both be "instanceof" the same interface type,...
- **How can I know when I reach the last record in a table, since JDBC doesn't provide an EOF method?**
Java:API:JDBC:ResultSets
Yusuf Dönmez, Mar 23, 2002
 You can use last() method of java.sql.ResultSet, if you make it scrollable. Joe Sam Shirah adds: You can also use isLast() as you are reading the...



Related Links

[JDBC Forum](#)
[Sun's JDBC pages](#)
[JDBC 2.0 optional package](#)

[Wish List](#)
[Features](#)
[About jGuru](#)
[Contact Us](#)

- **Problem with getDouble() We use the Java method getDouble() to get numeric values. Up to now it worked properly but suddenly we got false values, like 49066.429000000004 for the true value 49066.429....**
Java:API:JDBC:Data Types
Joe Sam Shirah PREMIUM, Mar 23, 2002
 First, let me assure you that something has changed, whether driver, database column definition, or database update. Even so, you were just lucky in the...

- **How can I correctly parse CSV (comma separated values) files? StringTokenizer doesn't seem to fit many conditions.**
Java:API:JDBC, Java:API:IO:Tokenizing, Java:API:IO:Regular expressions
Joe Sam Shirah PREMIUM, Mar 23, 2002
 Ian Darwin has two classes (CSV.java and CSVRE.java) to handle CSV files in his Java Cookbook, including a way with regular expressions. You can download...

- **Where can I find info, frameworks and example source for writing a JDBC driver?**
Java:API:JDBC:Drivers
Joe Sam Shirah PREMIUM, Mar 23, 2002
 There are several drivers with source available, like MM.MySQL, SimpleText Database, FreeTDS, and RmiJdbc. There is at least one free framework, the j...

- **Is any JDBC driver available that can access EDI messages?**
Java:API:JDBC:Design, Implementation, and Performance, Java:API:JDBC:Drivers
Laurent Mihalkovic PREMIUM, Mar 23, 2002
 I don't know about existing drivers, but JavaPro had a good article on writing your own driver. The example was a driver to access XML files. check it...

- **How can I create a custom RowSetMetaData object from scratch?**
Java:API:JDBC:2.0
Joe Sam Shirah PREMIUM, Feb 28, 2002
 One unfortunate aspect of RowSetMetaData for custom versions is that it is an interface. This means that implementations almost have to be proprietary....

- **How does a custom RowSetReader get called from a CachedRowSet?**
Java:API:JDBC:2.0
Joe Sam Shirah PREMIUM, Feb 28, 2002
 The Reader must be registered with the CachedRowSet using CachedRowSet.setReader (javax.sql.RowSetReader reader). Once that is done, a call to CachedR...

- **How do I implement a RowSetReader? I want to populate a CachedRowSet myself and the documents specify that a RowSetReader should be used. The single method accepts a RowSetInternal caller and returns...**
Java:API:JDBC:2.0
Joe Sam Shirah PREMIUM, Feb 28, 2002
 The documentation says "It can be implemented in a wide variety of ways..." and is pretty vague about what can actually be done. In general, readData()...

- **How can I instantiate and load a new CachedRowSet object from a non-JDBC source?**
Java:API:JDBC:2.0
Joe Sam Shirah PREMIUM, Feb 28, 2002
 The basics are: Create an object that implements javax.sql.RowSetReader, which loads the data. Instantiate a CachedRowset object. Set the...

- **Can I set up a connection pool with multiple user IDs? The single ID we are forced to use causes problems when debugging the DBMS.**
Java:API:JDBC:Connections
Joe Sam Shirah PREMIUM, Feb 26, 2002
 Since the Connection interface (and the underlying DBMS) requires a specific user and password, there's not much of a way around this in a pool. While...

- **How can I protect my database password ? I'm writing a client-side java application that will access a database over the internet. I have concerns about the security of the database passwords. The client...**
Java:API:JDBC:Design, Implementation, and Performance
Jay Meyer, Feb 26, 2002
 This is a very common question. I answered a similiar question at Remote database over internet + Java Swing app with JDBC != secure? Conclusion:...

- **Detecting Duplicate Keys I have a program that inserts rows in a table. My table has a column 'Name' that has a unique constraint. If the user attempts to insert a duplicate name into the table, I want...**
Java:API:JDBC:Exceptions and Warnings
JIA Java Italian Association PREMIUM, Feb 26, 2002
 A solution that is perfectly portable to all databases, is to execute a query for checking if that unique value is present before inserting the row. The...

- **What driver should I use for scalable Oracle JDBC applications?**
Java:API:JDBC:DBMS/Product Specific
Joe Sam Shirah PREMIUM, Feb 16, 2002
 Sun recommends using the thin (type 4) driver. On single processor machines to avoid JNI overhead. On multiple processor machines, especially...

- **Can you scroll a result set returned from a stored procedure? I am returning a result set from a stored procedure with type SQLRPGLE but once I reach the end of the result set it does not allow repositioning....**
Java:API:JDBC:ResultSets, Java:API:JDBC:Stored Procedures
Joe Sam Shirah PREMIUM, Feb 16, 2002
 A CallableStatement is no different than other Statements in regard to whether related ResultSets are scrollable. You should create the CallableStatement...

- **Driver memory problem I am using interclient driver to connect to interbase. My program does lots of inserts, updates, selects, etc. The problem I found is that if I create a statement for each operation...**
Java:API:JDBC:Statements
Joe Sam Shirah PREMIUM, Feb 16, 2002
 Regardless of DBMS, there's no particular reason to close a Statement until you are done with your operations - assuming 1) a JDBC compliant driver works...

- **How do I write Greek (or other non-ASCII/8859-1) characters to a database?**
Java:API:JDBC:Design, Implementation, and Performance
Joe Sam Shirah PREMIUM, Jan 31, 2002
 From the standard JDBC perspective, there is no difference between ASCII/8859-1 characters and those above 255 (hex FF). The reason for that is that...

- **How can I insert images into a Mysql database?**
Java:API:JDBC:DBMS/Product Specific
Kasi Mono, Jan 31, 2002
 This code snippet shows the basics: `File file = new File(fPICTURE); FileInputStream fis = new FileInputStream(file); PreparedStatement ps = ...`

- **I'd like to evaluate the Object/Relational Mapping approach and products. Any discussion is helpful.**
Java:API:JDBC:Design, Implementation, and Performance
Matt Goodall, Jan 31, 2002
 jguru Disclaimer: The following views and opinions are entirely those of the respondents and provided for our readers to consider. jGuru is not in the...

[<< previous](#)[beginning](#)[next >>](#)



Online Catalog

Sponsored by:

O'Reilly Home
Press Room
Jobs

Resource Centers

Perl
 Java
 Python
 C/C++
 Scripting
 Web
 Digital Media
 Web Services
 XML
 Oracle
 SysAdm/Networking
 Security
 Databases
 Linux/Unix
 Macintosh/OS X
 Windows
 .NET
 Open Source
 Wireless
 Bioinformatics

Book Series

Hacks
 Head First
 Cookbooks
 In a Nutshell
 CD Bookshelves
 Pocket References
 The Missing Manuals

Publishing Partners

No Starch Press
 Paraglyph Press
 Syngress Publishing



Online Publications

LinuxDevCenter.com
 MacDevCenter.com
 ONDotnet.com
 ONJava.com
 ONLamp.com
 OpenP2P.com
 Perl.com
 WebServices.XML.com
 XML.com

Special Interest

Events
 Meerkat News
 Ask Tim
 tim.oreilly.com
 From the Editors List
 Letters
 Beta Chapters
 Newsletters
 Open Books

Special Sales

Academic
 Corporate
 Government

Inside O'Reilly

About O'Reilly
 International
 Media Kit
 Contact Us
 Catalog Request


[see larger cover](#)

[Table of Contents](#)
[Index](#)
[Errata](#)
[Sample Chapter](#)
[Examples](#)
[Java Persistence Library](#)
[Colophon](#)

[Register your book](#) to get email notification of new editions, special offers, and more.

Database Programming with JDBC and Java

By [George Reese](#)

1st Edition June 1997

1-56592-270-0, Order Number: 2700

This book has been updated--the edition you're requesting is out of print. Please visit the catalog page of the [latest edition](#).

The latest edition is also available on [Safari Bookshelf](#).

Buy from O'Reilly: [View Cart](#)

This book describes the standard Java interfaces that make portable object-oriented access to relational databases possible and offers a robust model for writing applications that are easy to maintain. It introduces the JDBC and RMI packages and includes a set of patterns that separate the functions of the Java application and facilitate the growth and maintenance of an application. *Database Programming with JDBC and Java* recently won a Reader's Choice Special Mention Award from *Visual Basic Programmers Journal*. It was the only winner in the category of Java publications. [[Full Description](#)]

Sample Chapter and Code Examples

[Chapter 4: Database Access Through JDBC](#)

Download the code examples from this book. The complete set of examples is available at: <http://examples.oreilly.com/javadata/>

Related Books:

- [Developing Java Beans](#) (O'Reilly)
- [Java AWT Reference](#) (O'Reilly)
- [Java Cryptography](#) (O'Reilly)
- [Java Distributed Computing](#) (O'Reilly)
- [Java Examples in a Nutshell, 2nd Edition](#) (O'Reilly)
- [Java Fundamental Classes Reference](#) (O'Reilly)
- [Java in a Nutshell, 4th Edition](#) (O'Reilly)
- [Java Language Reference, 2nd Edition](#) (O'Reilly)
- [Java Network Programming](#) (O'Reilly)
- [Java Security, 2nd Edition](#) (O'Reilly)
- [Java Servlet Programming, 2nd Edition](#) (O'Reilly)
- [Java Threads, 2nd Edition](#) (O'Reilly)
- [Java Virtual Machine](#) (O'Reilly)
- [JavaScript: The Definitive Guide, 4th Edition](#) (O'Reilly)
- [Learning Java, 2nd Edition](#) (O'Reilly)
- [Netscape IFC in a Nutshell](#) (O'Reilly)

Pocket References	Pocket References
Compact, Comprehensive, Easy-to-use	Compact, Comprehensive, Easy-to-use
See them all at: pocketrefs.oreilly.com	See them all at: pocketrefs.oreilly.com

[O'Reilly Home](#) | [Privacy Policy](#)

© 2003, O'Reilly & Associates, Inc.
webmaster@oreilly.com

All trademarks and registered trademarks appearing on oreilly.com are the property of their respective owners.

User Groups

Writing for O'Reilly

How to Order

Bookstores

Traveling to
a tech show?

[New York City Hotels](#)

[Seattle Hotels](#)

[Discount Hotels](#)

[Florida Hotels](#)

[BC Hotels](#)

[Los Angeles Hotels](#)



Go to Mobilized Software

Where the mobile experts meet


[HOME](#) | [ABOUT US](#) | [SUBSCRIBE TO DDJ](#) | [ADVERTISE WITH DDJ](#)

GO TO...

Microsoft®
Smart Client®

**Click here
to register today!**

[register now](#)

security
productivity
reliability

Microsoft
for Partners

[Printer Friendly Version](#)

SQL Access Group's Call-Level Interface

An independent interface for database development

Roger Sippl

Roger is chairman of the SQL Access Group and founder of Visigenic Software. He can be contacted at Visigenic Software, 951 Mariner's Island Blvd., Suite 460, San Mateo, CA 94404, (415) 286-1900.

The SQL Access Group (SAG) was formed in 1989 to define and promote standards for database portability and interoperability. Initially, the membership roster included database heavyweights Oracle, Informix, and Ingres (now Computer Associates), as well as hardware vendors Digital, Hewlett-Packard, Tandem, and Sun. [Table 1](#) lists the current membership roster. The group's initial projects involved developing the draft ISO/ANSI standard for SQL (including the embedded-SQL interface specifications) and a specification for remote data access (RDA).

In 1990, SAG took the lead in developing an SQL-based Call Level Interface (CLI). The CLI SAG is an API for database access, offering an alternative invocation technique to embedded SQL that provides essentially equivalent operations. SAG envisioned an interface that would enable client/server applications to access data stored in heterogeneous relational and nonrelational databases. The interface would be platform, vendor, database, and language neutral. SAG and X/Open published the CLI Snapshot Specification in 1992 as a "work in progress," and it was adopted for use in commercial software products.

Microsoft helped define the X/Open CLI specification and became the first company to commercialize the CLI specification by shipping Open Database Connectivity (ODBC) 1.0 for Windows in 1992. To create ODBC, Microsoft extended the CLI specification and created a three-layer specification in which the "core" layer corresponds to the SAG CLI. Over the next two years, the CLI specification underwent several transformations, reemerging in 1994 as an X/Open Preliminary Specification. Also in 1994, Microsoft released ODBC 2.0, whose core functionality was still aligned with the SAG CLI. Earlier this year, Microsoft announced that ODBC 3.0 (to be released in 1996) will be fully aligned with both ISO's CLI standard and SAG's CLI Specification.

In March 1995, the CLI was finalized and published as an X/Open Common Application Environment (CAE) specification. CAE specifications are adopted by consensus and are the basis against which suppliers brand their products. The adoption and publication of the CLI as an X/Open CAE specification represents the culmination of five years of cooperative development work within the SQL Access Group.

In the meantime, ODBC has gained broad support in the database industry. All major software vendors--Oracle, Sybase, Informix, Computer Associates, IBM, Gupta, Powersoft, and Borland, as well as more than 140 application-software developers and VARs--have added ODBC support. Many of these vendors also offer proprietary APIs; nonetheless, they see the X/Open CLI specification and ODBC as important to their strategy. In 1994, Microsoft granted an exclusive source-code license to Visigenic Software for porting and licensing the ODBC SDK to non-Windows platforms. As a result, the ODBC SDK is now also available on all major UNIX platforms, as well as OS/2 and Macintosh.

A Closer Look at the SAG CLI

The X/Open CLI specification is a standard API for database access that is vendor, platform, and database neutral. It defines a set

**Go to
Mobilized
Software:**

**The Ultimate
Source for Mobile
and Wireless
Applications**



of functions that a program can call directly using normal function-call facilities. The specification is language independent and includes header files for both C and Cobol.

The CLI specification defines 57 functions that support a rich set of database-access operations sufficient for creating robust database applications, including:

- Allocating and deallocating handles (eight calls).
- Getting and setting attributes (ten calls).
- Opening and closing database connections (two calls).
- Accessing descriptors (six calls).
- Executing SQL statements (nine calls).
- Retrieving results (eight calls).
- Accessing schema metadata (four calls).
- Performing introspection (four calls).
- Controlling transactions (two calls).
- Accessing diagnostic information (three calls).
- Canceling functions (one call).

A database application calls these functions for all interactions with a database. The CLI enables applications to establish multiple database connections simultaneously and to process multiple statements simultaneously, depending on the capabilities of the database servers being accessed. [Figure 1](#) and [Figure 2](#) show the basic control flow for using the CLI functions.

The X/Open CLI specification was developed with client/server architectures in mind. In fact, the CLI is ideal for this environment, in which the developer often knows little (if anything) about the database at the time the application is written. The X/Open CLI specification defines a set of introspection functions that enable an application to discover the characteristics and capabilities of a particular CLI implementation and of any database server accessed through that implementation. For example, *SQLGetTypeInfo* lets you find out what data types are supported by a particular server, and *SQLDataSources* returns a list of available database servers and descriptions. Introspection functions facilitate a technique known as "adaptive programming," whereby an application adapts its behavior at run time to take advantage of the capabilities of a particular database environment.

Processing SQL Statements

The X/Open CLI specification, including sample programs and header files, is available directly from X/Open. Listings [One](#), [Two](#), and [Three](#) illustrate how the CLI works. [Listing One](#) is from a typical database application. Listings [Two](#) and [Three](#) are the significant portions of functions called by [Listing One](#).

The application allocates memory for an environment handle and a connection handle; both are required to establish a database connection. The *SQLConnect* call establishes the database connection, specifying the server name (*server_name*), user id (*uid*), and password (*pwd*). The application then allocates memory for a statement handle and calls *SQLExecDirect*, which both prepares and executes an SQL statement. The *SQLGetDiagField* call takes advantage of the CLI's ability to interrogate the database server. In this case, it returns a code describing the nature of the SQL statement just executed. With this information in hand, the application calls the user-defined *DoSelect()* function to process the results of the SQL statement. Finally, the program frees the statement handle, disconnects from the database, and frees the memory previously allocated for the connection and environment handles.

The body of the *DoStatement()* function in [Listing Two](#) is built around a switch statement that processes the results of an SQL statement based on the return value from the *SQLGetDiagField* call in [Listing One](#). In the case of a SELECT statement, which requires its own complex processing, the function calls another user-defined function, *DoSelect()*; see [Listing Three](#). In the case of an UPDATE, DELETE, or INSERT statement, the *DoStatement()* function calls the CLI diagnostic function *SQLGetDiagField* to find out how many rows were affected by the statement, then calls *SQLEndTran* to commit the transaction. The function prints one message indicating whether the commit was successful and another giving the number of affected rows.

In the case of any Data Definition Language (DDL) statement, the *DoStatement()* function first calls the CLI introspection function *SQLGetInfo* to find out if the CLI implementation being used supports transaction processing for DDL statements. If so, the function calls *SQLEndTran* to commit the transaction, then prints a message indicating whether the commit was successful.

The *DoSelect()* function in [Listing Three](#) processes the results of an SQL SELECT statement. It is called by the *DoStatement()* function. First, *DoSelect()* calls *SQLNumResultCols* to determine how many columns are in the result set. Then, for each column in the result set, the function calls *SQLDescribeCol* to get descriptive information about the column (that is, its length, scale, and data type), prints an appropriate row of column headings to the standard output, allocates memory to bind the results, and calls *SQLBindCol* to establish the bindings. Next, *DoSelect()* calls *SQLFetch* to fetch rows from the result set until none are left. For each row, the *DoSelect()* function prints the column values, followed by a new line. The CLI can provide various types of diagnostic information, such as whether a value is truncated or null; *DoSelect()* tests for these two conditions and, when they occur, calls the user-defined *BuildMessage()* function (not shown) to generate appropriate error messages. At the end, the function prints a list of any such error messages. Finally, the application closes the cursor for the statement handle and frees the data buffers.

What's Next?

The SQL Access Group formally merged with X/Open in 1995. The charters of SAG and X/Open's Data Management Working

Group were essentially the same, and X/Open had always edited and published the work of the SQL Access Group. The merger made it possible to eliminate duplicate efforts, reduce costs, and unify development efforts. The X/Open Data Management Technical Committee disbanded, and the X/Open SQL Access Group, now functioning within the X/Open Technical Program, has assumed all of its responsibilities.

SAG is working in close cooperation with ISO on its upcoming SQL CLI specification, which is intended to mirror the X/Open specification. SAG is also actively pursuing the next logical step for the CLI--the development of an X/Open test suite for CLI conformance. Such a test suite will enable developers to verify conformance to the CLI Specification. Development of the test suite should be well underway by the time this article is published.

SAG has already begun work on the next version of the CLI specification. Our mission is to extend and refine the CLI for even more successful interoperability and to define standards that incorporate newer database technologies, including XA transaction processing, stored procedures, BLOBs, triggers, and asynchronous calls.

The efforts of the major standards organizations, including ISO, ANSI, and X/Open, as well as the strategies of all the major players in the database industry, now incorporate the X/Open CLI Specification. Vendors and standards organizations are moving rapidly in the same direction, a direction the marketplace has already validated.

Figure 1: Basic CLI control flow.

Figure 2: CLI control flow for processing SQL statements.

Table 1: SQL Access Group members.

AT&T
 Borland International
 Computer Associates
 Fulcrum Technologies
 Hitachi
 IBM
 Information Builders
 Informix Software
 INTERSOLV
 Microsoft
 Oracle
 Progress Software
 Sybase
 Visigenic Software

Listing One

```
/* allocate an environment handle */
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
/* allocate a connection handle */
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
/* connect to database */
if (SQLConnect(hdbc, server_name, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS) !=
    SQL_SUCCESS)
    return(PrintErr(SQL_HANDLE_DBC, hdbc));
/* allocate a statement handle */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
/* execute the SQL statement */
if (SQLExecDirect(hstmt, sqlstr, SQL_NTS) != SQL_SUCCESS)
    return(PrintErr(SQL_HANDLE_STMT, hstmt));
/* see what kind of statement it was */
SQLGetDiagField(SQL_HANDLE_STMT, hstmt, 0, SQL_DIAG_DYNAMIC_FUNCTION_CODE,
    (SQLPOINTER)&stmttype, 0, (SQLSMALLINT *)NULL);
/* process the SQL statement */
DoStatement(stmttype);
/* free statement handle */
SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
/* disconnect from database */
SQLDisconnect(hdbc);
/* free connection handle */
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
/* free environment handle */
SQLFreeHandle(SQL_HANDLE_ENV, henv);
```

Listing Two

```
switch(stmttype) {
```

```

/* SELECT statement */
case SQL_DIAG_SELECT_CURSOR:
    DoSelect();
    break;
/* searched UPDATE, searched DELETE, or INSERT statement */
case SQL_DIAG_UPDATE_WHERE:
case SQL_DIAG_DELETE_WHERE:
case SQL_DIAG_INSERT:
    /* get row count */
    SQLGetDiagField(SQL_HANDLE_STMT, hstmt, 0, SQL_DIAG_ROW_COUNT,
        (SQL_POINTER *)&rowcount, 0, (SQLSMALLINT *)NULL);
    if (SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT) == SQL_SUCCESS)
        printf("Operation successful\n");
    else
        printf("Operation failed\n");
    printf("%ld rows affected\n", rowcount);
    break;
/* other statements */
case SQL_DIAG_ALTER_TABLE:
case SQL_DIAG_CREATE_INDEX:
case SQL_DIAG_CREATE_TABLE:
case SQL_DIAG_CREATE_VIEW:
case SQL_DIAG_DROP_INDEX:
case SQL_DIAG_DROP_TABLE:
case SQL_DIAG_DROP_VIEW:
case SQL_DIAG_GRANT:
case SQL_DIAG_REVOKE:
    SQLGetInfo(hdbc, SQL_TXN_CAPABLE, &txn_type, 0, 0);
    if(txn_type == SQL_TC_ALL) {
        if (SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT) ==
            SQL_SUCCESS)
            printf("Operation successful\n");
        else
            printf("Operation failed\n");
    }
    break;
/* other implementation-defined statements */
default:
    printf("Statement type=%ld\n", stmttype);
    break;
}

```

Listing Three

```

/* determine number of result columns */
SQLNumResultCols(hstmt, &nresultcols);
/* display column names */
for (i=0; i<nresultcols; i++) {
    SQLDescribeCol(hstmt, i+1, colname, sizeof(colname), &colnamelen,
        &coltype, collen[i], &scale, &nullable);
    /* user-defined function to get the display length for the data type */
    collen[i] = DisplayLength(coltype, collen[i], colname);
    printf("%*.s", collen[i], collen[i], colname);
    /* allocate memory to bind column */
    data[i] = (SQLCHAR *) malloc(collen[i]+1);
    /* bind columns to program vars, converting all types to CHAR */
    SQLBindCol(hstmt, i+1, SQL_CHAR, data[i], collen[i]+1, &outlen[i]);
}
printf("\n");
/*display result rows */
while (SQL_SUCCEEDED(rc=SQLFetch(hstmt))) {
    errmsg[0] = '\0';
    for (i=0; i<nresultcols; i++) {
        if (outlen[i] == SQL_NULL_DATA || outlen[i] >= collen[i])
            /* set data text to "NULL" or add to errmsg */
            BuildMessage(errmsg, (SQLPOINTER *)&data[i], collen[i],
                &outlen[i], i);
        printf("%*,s ", outlen[i], outlen[i], data[i]);
    } /* for all columns in this row */
    /* print any accumulated error messages and new line */
    printf("%s\n", errmsg);
} /* while rows to fetch */
SQLCloseCursor(hstmt);

```

```

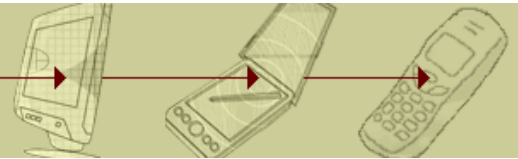
/* free data buffers */
for (i=0; i<nresultcols; i++) {
    free(data[i]);
}
End Listings

```

[HOME](#) | [ABOUT US](#) | [SUBSCRIBE TO DDJ](#) | [ADVERTISE WITH DDJ](#)

Advertisement

Go to Mobilized Software



MarketPlace

[Ektron - Web Content Management - 30 day Trial](#)

Facing fewer resources and heavier workloads? Download a Free copy of Ektron's Web CMS. Become part of the 9,300 other organizations that use Ektron solutions to efficiently manage their web content.

[IntelliVIEW - Interactive Reporting Tool](#)

Easy-to-use XML based Reporting solution. Create any kind of report. Access virtually any data source. Manipulate data using drag & drop facilities. Slash development time for creating reports by over 75%. Publish Reports easily over the web. No 'Per-client' Licensing Fees. Supports Java & .NET

[Bug Tracking Like You've Never Seen Before](#)

Full API hooks ExtraView bug tracking into your applications, Web, source control & testing tools.

[Need Charting Muscle?](#)

Engineering, Scientific, Financial, and Serious Business Charting components fit seamlessly into your EXE or Web Site. Includes WinForm, WebForm, ActiveX, and DLL interfaces. GigaSoft ProEssentials v5

[GoToMyPC: Secure Enterprise Remote Access](#)

All the TCO advantages of a managed service with multi-level security and control. Click here for more information and a complimentary evaluation of GoToMyPC.

[Wanna see your ad here?](#)

SPONSORED LINKS

Travel: [Hotels](#) • [Discount Travel](#) • [US Hotels](#) • [Las Vegas Hotels](#) • [Popular Hotels](#) | Finance: [Debt Consolidation](#) • [Personal Loans](#) | Electronics [Digital Cameras](#) • [Digital Camera](#) • [Sony Digital Camera](#) • [Camcorders](#) • [Inkjet Cartridges - Printer Ink Cartridges](#) | Miscellaneous: [Cheap Computers](#) • [Hand Tools : Power Tools](#) • [Systems Management Software](#) • [SEO Company](#) • [Dictionary](#) • [Furniture Auctions - Handbags Auction](#)

Copyright © 2003 CMP Media LLC, Dr. Dobb's Journal's [Privacy Policy](#), Comments: webmaster@ddj.com

SDMG Websites: [BYTE.com](#), [C/C++ Users Journal](#), [Dr. Dobb's Journal](#), [MSDN Magazine](#), [Sys Admin](#), [SD Expo](#), [SD Magazine](#), [Unixreview](#), [Windows Developer Network](#), [New Architect](#)

web1



The Source for Java Developers



[Developers Home](#) > [Products & Technologies](#) > [Java Technology](#) > [J2SE](#) > [Desktop Java](#) > [JavaBeans](#) > [Learning](#) > [Tutorial and Code Camps](#) > [Introducing Java Beans](#) >

[Profile and Registration](#) | [Why Register?](#)

Tutorials & Code Camps

Java Beans, Pt. 1: Definition: What is a Bean?

[Printable Page](#)

[Training Index](#)

Definition: What is a Bean?

by Greg Voss

[\[Introducing Java Beans\]](#) [\[Reusable Software Components\]](#)

[\[Application Builder Tools\]](#) [\[Basic Bean Concepts\]](#)

If you have used Delphi, or Visual Basic, you are already familiar with the notion of a bean. The idea is the same; the programming language is different. A Java Bean is a reusable software component that works with Java. More specifically: a Java Bean is a reusable software component that can be visually manipulated in builder tools.

Definition:

A Java Bean is a reusable software component that can be visually manipulated in builder tools.

To understand the precise meaning of this definition of a Bean, clarification is required for the following terms:

- Software component
- Builder tool
- Visual manipulation

Each of these will be addressed in turn.



[Company Info](#) | [About SDN](#) | [Press](#) | [Contact Us](#) | [Employment](#)
[How to Buy](#) | [Licensing](#) | [Terms of Use](#) | [Privacy](#) | [Trademarks](#)

Copyright 1994-2004 Sun Microsystems, Inc.

A Sun Developer Network Site

Unless otherwise licensed, code in all technical manuals herein (including articles, FAQs, samples) is provided under this [License](#).

[Content Feeds](#)

▲ e-Business Engineering Vorlesung

▼ 1 Motivation und Einführung

▼ 1.1 Was ist e-Business?

▼ 1.2 Relevante Techniken und ihre Einordnung

▼ 1.3 Schichtenmodell moderner e-Business Applikationen

▼ 2 Datenhaltung und Datenzugriff

▼ 2.1 Extensible Markup Language (XML)

▼ • Strukturelle Grundkonzepte

▼ • XML Namensräume

▼ • XML Schema

▼ • Die Lokatorsprache XPath

▼ 2.2 Java Database Connectivity (JDBC)

▼ 3 Anwendungslogik und serverinterne Kommunikation

▼ 3.1 Java Remote Method Invocation (RMI)

▼ 3.2 Servlets

▼ 3.3 Enterprise Java Beans

▼ 4 Server-seitige Präsentation und Sicherheitsaspekte

▼ 4.1 Java Server Pages (JSP)

▼ 4.3 XML Transformationen mit XSLT

▶ Hinweise zum Scriptum

▶ Empfohlene Literatur

▲ 1 Motivation und Einführung

1.1 Was ist e-Business?

Der Begriff des *e-Business* als Abkürzung des englischsprachigen *electronic Business* hat sich inzwischen als Subsumption aller für ein Unternehmen wertschöpfenden Aktivitäten im Internet eingebürgert.

Die Sinngabe greift damit weiter als der historisch ältere Begriff *e-Commerce*, welcher ursprünglich ausschließlich Verkaufsaktivitäten bezeichnete. Inzwischen werden beide Terme jedoch nahezu synonym verwendet. Teilweise findet sich für den Teilbereich des internetgestützten Verkaufs von Waren und Dienstleistungen an Endkunden auch die Bezeichnung *e-tailing* (für *electronic retailing*) welcher jedoch nur einen Teilaspekt des e-Commercebegriffes abzudecken vermag.



Definition 1: e-Business

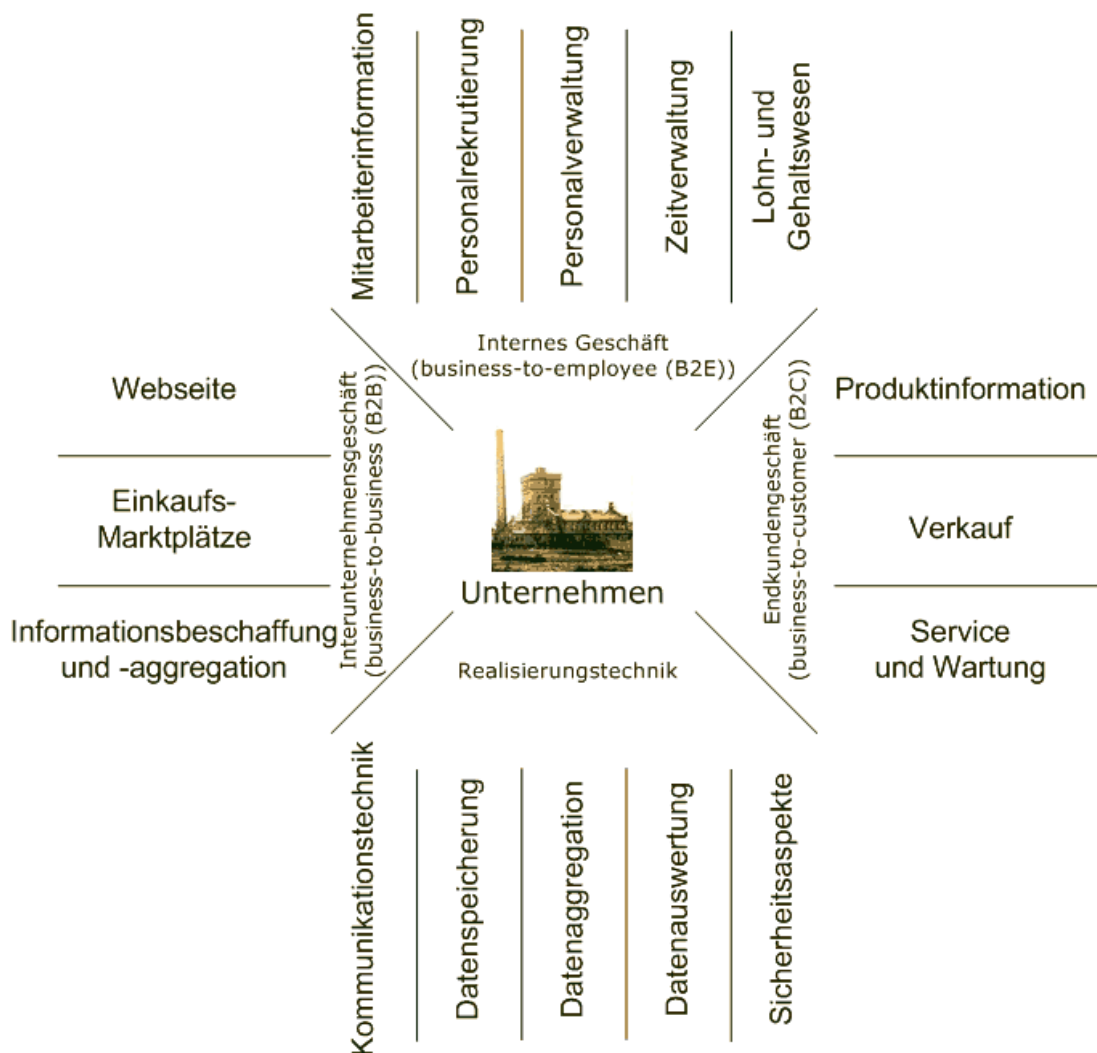
Electronic Business ist die Gesamtheit aller unternehmerischen Aktivitäten im Internet.

Gemäß dieser allgemeinen Definition werden sämtliche auf das Unternehmensziel gerichtete nach außen wirkende Aktivitäten als e-Business eingeordnet.

Gleichzeitig ergibt sich aus der Abstützung auf der Realisierungstechnik des Internets auch eine interne Sichtweise, sobald diese Technik innerhalb des Unternehmens zum Einsatz kommt.

Die Darstellung der [Abbildung 1](#) unternimmt den Versuch der Einordnung der sich ergebenden Anwendungsdimensionen des e-Businessbegriffs.

Abbildung 1: Dimensionen des e-Business



Naheliegendste Form des e-Business ist der Geschäftsverkehr mit dem (End-)Kunden, als dem typischen Konsumenten der durch ein Unternehmen zur Verfügung gestellten Güter und Dienstleistungen. Dieser Teilbereich wird mit dem Begriff *Business-to-Customer* (B2C) belegt. In diese e-Businessvariante fallen alle Interaktionen zwischen Kunde und Unternehmen während des gesamten Lebenszyklus des angebotenen Produkts, angefangen von verkaufsfördernden Maßnahmen (Marketing) über den Verkaufs- bzw. Dienstleistungserbringungsakt selbst bis hin zur Abwicklung der Wartung, soweit nach Art des angebotenen Gutes elektronisch überhaupt möglich.

Entgegengesetzt zum durch ein Unternehmen produzierten ausgehenden Güter- und Dienstleistungsstrom verläuft die Beschaffung von nicht-menschlichen Produktionsfaktoren wie Roh-, Hilfs- und Betriebsstoffen sowie die Interunternehmenskommunikation. Dieser Teilbereich wird mit dem Begriff *Business-to-Business* (B2B) belegt. In diese e-Businessvariante fallen die zwischen Unternehmungen ablaufenden elektronischen Kommunikationen. Die Spannbreite reicht hierbei von der kostenfrei nutzbaren statischen Präsentation des Güter- und Dienstleistungsangebots im Stile eines Katalogs über spezialisierte Marktplätze mit Angebots- und Nachfragefunktionalitäten bis hin zu Informationsdienstleistungen welche Zugriff auf die datenhaltenen Systeme des Geschäftspartners gewähren.

Die umfassende Betrachtung der zuvor ausgeklammerten Kommunikation mit potentiellen und bestehenden Mitarbeitern konstituiert die dritte Klasse der e-Businessanwendungen, welche auf die unternehmensinterne Kommunikation mit den Mitarbeitern fokussieren. Dieser Teilbereich wird mit dem Begriff *Business-to-Employee* (B2E) belegt.

Dieser Sparte werden alle elektronischen Informationsangebote an den Mitarbeiter, wie Auskunft über den aktuellen Gleitzeitstand, Adressstamm- sowie Gehaltsdaten, zugeordnet.

1.2 Relevante Techniken und ihre Einordnung

Orthogonal zu den drei Anwendungsdimensionen verdient die ebenfalls in [Abbildung 1](#) dargestellte Realisierungstechnik Betrachtung.

Hierunter fallen gemäß [Definition 1](#) alle sog. *Internettechniken*.

Dieser, in der Praxis nicht klar definiert und trennscharf gebrauchte Begriff umfaßt sowohl die Internetbasistechniken zur Datendarstellung und -übertragung als auch verschiedene Techniken zur Realisierung von Anwendungen, die über das Internet angesprochen und benutzt werden können.

Im wesentlichen zielen die eingesetzten Techniken auf die Lösung spezifischer Problemstellungen. [Tabelle 1](#) stellt die im Rahmen der Vorlesung behandelten Techniken nebst den durch sie betrachteten Problemgebieten und einer Kurzcharakteristik zusammen.

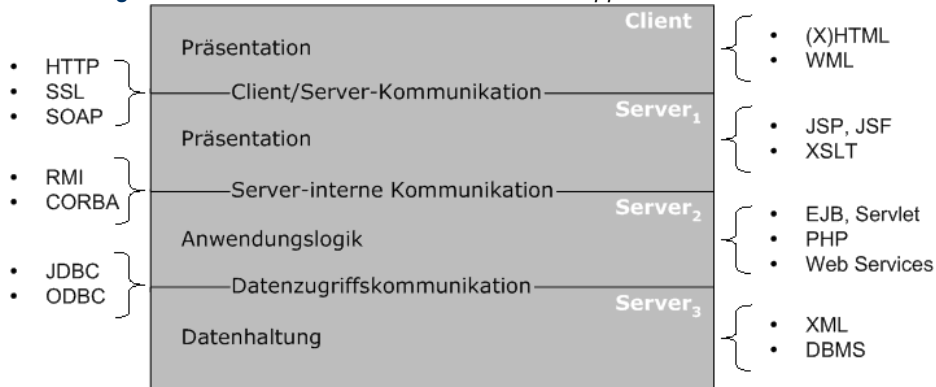
Tabelle 1: Techniken: Einordnung und Kurzcharakterisierung

Problemdomäne	Technik	Charakteristik
Anwenderpräsentation	HTML	Einfache Hypertextsprache zur Darstellung textbasierter Information durch einen (Internet-)Browser .
	WML	Hypertextsprache zur Darstellung textueller Informationen auf ressourcenbeschränkten mobilen Geräten.
Systemneutrale Datendarstellung	XML	Generische Auszeichnungssprache zur Darstellung beliebiger Daten.
Serverseitige Verarbeitung	Enterprise Java Bean	Durch SUN erarbeitete Komponententechnik.
	Servlet	Durch SUN erarbeiteter Ansatz zur Funktionalitätserweiterung eines Web Servers.
	Java Server Page	Durch SUN erarbeiteter Ansatz zur dynamischen serverseitigen Erzeugung von Webseiten.
	Java Server Faces	Durch SUN erarbeiteter Ansatz zur vereinfachten Erstellung von GUI-basierten Web-Dialoganwendungen.
	Die Extensible Stylesheet Language XSLT	W3C-Standard zur Transformation von XML-Inhalten. (Auch clientseitige Ausführung möglich)
Datenbankzugriff	Java Database Connectivity	Durch SUN erarbeiteter Ansatz für den Zugriff auf tabellenartige Datenquellen. Zumeist für den Zugriff auf relationale Datenbanken benutzt.
Sicherheitsaspekte	Digitale Signatur	Mechanismus zur Sicherstellung von Berechtigung, Glaubwürdigkeit des Ursprungs, Verbindlichkeit und (Daten-)Konsistenz.
	Verschlüsselung	Mechanismus zur Sicherstellung der Vertraulichkeit.
	Leitungssicherheit	Mechanismus zur für die verwendende Applikation transparenten Sicherung einer unsicheren Verbindung.
Kommunikation	HyperText Transport Protocol	Basisprotokoll des WWW.
	Web Services/SOAP	Anwendung der XML zur Realisierung von Nachrichtenaustausch und entfernten Funktionsaufrufen.
	RMI	Durch SUN für Java adaptierte Variante entfernter Funktionsaufrufe.

1.3 Schichtenmodell moderner e-Business Applikationen

Abbildung 2 ordnet die zuvor eingeführten Techniken in ein Schichtenmodell für e-Business Applikationen ein.

Abbildung 2: Schichtenmodell moderner e-Business Applikationen



(click on image to enlarge!)

Das Modell unterscheidet vier Anwendungsschichten die jeweils durch Kommunikationsbeziehungen miteinander verknüpft sind.

Die **Client-Präsentationsschicht** dient der Ausgabe von Informationen an den Endanwender und im Bedarfsfalle der Interaktion mit ihm.

Die **serverseitige Präsentationsschicht** dient der Erzeugung der an den Client versandten Präsentation. Innerhalb dieser Schicht können anwender- oder ausgabegerätespezifische Anpassungen (Personalisierung) der serverseitig erzeugten Ausgabe vorgenommen werden bevor sie über das Netzwerk gesendet wird.

Die Systemfunktionalität (die sog. Business Logik) wird durch die **Anwendungslogikschicht** realisiert.

Persistente Datenspeicher werden durch die **Datenhaltungsschicht** zur Verfügung gestellt.

An den Schnittstellen der verschiedenen Schichten werden unterschiedliche Kommunikationsstrategien und -protokolle eingesetzt, die speziell für den jeweiligen Anwendungsfall angepaßt sind.

Das logische Modell moderner e-Business Applikationen trifft keine Aussagen über die physische Verteilungssituation einer resultierenden Anwendung. So kann jede Schicht auf einer physische separierten Rechneinheit zur Ausführung gebracht werden, ebenso ist die Installation auf genau einem Rechner denkbar.

▲ 2 Datenhaltung und Datenzugriff

2.1 Extensible Markup Language (XML)

Einführung

Im Grunde besitzt die Geschichte der eXtensible Markup Language zwei Anfänge. Einerseits stellt XML die evolutionäre Fortentwicklung existierender generischer Auszeichnungssprachen dar; andererseits sind die Hintergründe der Sprache XML so eng mit dem Aufkommen des World Wide Webs (WWW) verwoben, daß die Geschichte auch hier ihren Anfang nehmen könnte...

Der chronologischen Ordnung folgend sei zunächst die Entwicklung aus der Idee des *Hypertext* aufgerissen.

Die ersten Ideen zum Konzept des Hypertexts, als Plan zur Überwindung der Beschränkungen und Unzulänglichkeiten des klassischen textbasierten Publikationsmediums Papier, datieren zurück bis in die 1950er Jahre. Sie postulieren neben der nichtsequentiellen Organisation des Mediums auch zentrale Begriffe wie *Knoten*, *Link*, *Anker* und *Netz*. Ziel dieser Überlegungen war es, den auszudrückenden Inhalt von editorielle- und Präsentationsinformation wie Seitenzahlen, Fußnoten, Paginierung usw. zu trennen. Durch die nichtlineare Organisation soll es dem Leser freigestellt werden, auf welchen Pfaden er sich durch das Dokument bewegt.

Zur Realisierung dieser Bemühungen wird das Dokument mit weiteren Informationen angereichert, die jedoch für den Leser unsichtbar bleiben. Dieser Gedanke reicht zurück bis in die Anfänge des Buchdrucks. Dort sind formatierungsorientierte Auszeichnungssymbole, etwa für Fettdruck oder Unterstreichungen, seit jeher bekannt. Vor dem Aufkommen der *what you see is what you get* Textverarbeitungssysteme waren diese bildlichen Symbole die einzige Möglichkeit zur Kommunikation präsentationsorientierter Information an den Schriftsetzer und Drucker. Jedem Schüler ist bereits ein weiteres Beispiel einer editoriiellen Auszeichnungssprache bekannt: Die graphischen Korrekturzeichen der Deutschlehrer. Auch sie liefern Informationen über den Inhalt, die nicht Bestandteil des Dokuments sind.

Voraussetzung für die angestrebte Flexibilisierung der Struktur eines Textes ist eine -- wie auch immer geartete -- technische Unterstützung. Seit den 60er Jahren wurden hierfür die aufkommenden elektronischen Rechanlagen herangezogen. Eine der ersten Aktivitäten hierzu ist das von [Ted Nelson](#) initiierte (inzwischen legendäre) [Xanadu-Projekt](#).

Zunächst erforderte die maschinelle Verarbeitung die Überarbeitung des Auszeichnungssymbolvorrates. Dies wurde notwendig, da eingesetzte Technik keine Unterstützung der alt-hergebrachten graphischen Auszeichnungssymbole bot.

In einem ersten Entwicklungsschritt wurden daher die vormalig bildhaften Zeichen durch textuelle Pendanten ersetzt und verallgemeinert. Beispielsweise: Überschrift zur inhaltlichen Kennzeichnung einer entsprechenden Textzeile.

Mit diesem Schritt erfolgte auch der Übergang zur formatierungsunabhängigen Auszeichnung, die bewußt auf die Beschreibung des späteren visuellen Aussehens der Information zugunsten einer neutralen deskriptiven Beschreibung der Semantik verzichtete.

In den 60er und 70er Jahren werden verschiedene Weiterentwicklungen der generischen Auszeichnungssprachen betrieben; u.a. bei der IBM durch das Team um Goldfarb, Mosher und Loire. Sie stellen 1969 unter dem Namen *Generalized Markup Language* einen Sprachvorschlag zusammen, der in der Folgezeit durch IBM kommerziell vermarktet wird.

Aus den GML-Aktivitäten bei IBM entwickelt sich die internationale Standardisierungsbewegung der *Standard GML* (SGML).

Durch sie wird eine Sprache festgelegt, welche die Definition eigener Sprachen erlaubt; daher auch der Begriff *Metasprache*. SGML bietet somit keinen feststehenden problemspezifischen Sprachumfang an, sondern eine Menge verschiedenster struktureller Konstrukte zur Formulierung von Dokumentgrammatiken.

In der Praxis wird der Einsatz einer mit Hilfe von SGML definierten Sprache oftmals plakativ zum *Einsatz von SGML* verkürzt, obwohl diese Begrifflichkeit lediglich den Erstellungsprozeß der Grammatik bezeichnet.

Mittels SGML definiert Tim Berners-Lee Mitte der 80er Jahre eine eigene Sprache zur vereinfachten Formulierung von Dokumenten, die er *HyperText Markup Language* (HTML) nennt.

Hauptbeweggrund seiner Aktivitäten ist der Versuch den Dokumentenaustausch am Europäischen Kernforschungszentrum CERN rechnergestützt zu vereinfachen.

Die Eingangs erwähnten zentralen Hypertextkonzepte finden sich bereits in seinem ersten Sprachvorschlag wieder. Zur technischen Realisierung der Verknüpfung zwischen den Dokumenten mittels Ankern und Links definiert er den *Uniform Resource Locator* (URL), eine global eindeutige Adresse für beliebige Inhalte.

Seine Aktivitäten in Genf bilden die Keimzelle des Web.

In der Folgezeit, insbesondere im Zuge der Kommerzialisierung des World Wide Web, entstehen verschiedene Revisionen der ursprünglichen HTML. Einige der Erweiterungen werden durch die beiden großen Web Browser Hersteller Microsoft und Netscape proprietär vorgenommen, um ihre Position am Markt zu stärken.

In der Konsequenz entstehen während des oft apostrophierten *browser war* teilweise inkompatible HTML-Dialekte. (Man denke nur an die Tags: *marquee* (nur Microsoft Internet Explorer) oder *layer* (nur Netscape Navigator))

Darüberhinaus entwickelt sich HTML zunehmend von einer Präsentations-orientierten Auszeichnungssprache zu einer semantischen. Dies bedeutet: während HTML in der ersten Grundform zunächst überwiegend Elemente bot, durch die die Präsentation der Inhalte am Bildschirm festgelegt wurde (Beispiele: *b* für Fettdruck, *u* für Unterstreichungen oder *i* für Kursivschreibung), wurden später zunehmend semantische Elemente eingeführt. Durch sie wird die Bedeutung der ausgezeichneten Information ausgedrückt (Beispiele hierfür: *acronym* zur Kennzeichnung von Abkürzungen, *address* für Adressen oder *strong* zur besonderen Betonung einer Textpassage).

So wünschenswert die sukzessive Umgestaltung der HTML an die veränderten Bedürfnisse war, so

aussichtslos waren die Bemühungen dennoch. Während bei den Präsentations-orientierten Elementen zunehmend Vollständigkeit hinsichtlich der Anwenderwünsche erzielt werden konnte, offenbarten sich die bisher erfolgten semantischen Erweiterungen als permanent inadäquat. Letztlich war der Versuch, durch Standardisierung, semantische Erweiterungen in HTML einzubringen in doppelter Hinsicht zum Scheitern verurteilt:

1. birgt der Ansatz die Gefahr, die Elementmenge in unbekannte Größen zu erweitern
2. muß die Semantik jedes Tags definiert, abgestimmt und verabschiedet werden.

Aus diesen Gründen wurde seitens des W3C nach einer tragfähigeren Lösung gesucht. Unter Rückgriff auf die HTML-Wurzeln (als Anwendung der Metasprache SGML) wurde das Projekt *SGML for the Web* initiiert.

Der letztendlich verabschiedete Vorschlag zur *eXtensible Markup Language* (XML) bildet konzeptionell eine Untermenge der Sprachmöglichkeiten von SGML. Konsequenterweise ist jedes XML-Dokument auch ein gültiges SGML-Dokument.

Die Abweichung zu SGML wird besonders aus den Entwicklungszielen für XML deutlich:

1. Einfache Nutzung im Internet.
In Abkehr von den Hauptnutzung SGMLs als offline Dokumentationsformat wird die Untermengenbildung *XML* für die primäre Nutzung im Internet vorgenommen.
2. Unterstützung eines breiten Anwendungsspektrums.
Auch hier soll die Untermengenbildung das Einsatzspektrum über die Hauptnutzung SGMLs als Format der technischen Dokumentation hinaus befördern.
3. SGML Kompatibilität.
XML bildet eine echte Untermenge des ISO-Standards SGML, durch diesen Schritt kann jedes XML-Dokument auch als gültiges SGML-Dokument interpretiert und durch die entsprechenden SGML-Werkzeuge verarbeitet werden.
4. Einfache Applikationsentwicklung.
Die Untermengenbildung wird im Hinblick auf eine gegenüber SGML deutlich vereinfachte Entwicklung von XML verarbeitenden Applikationen vorgenommen.
5. Minimierung optionaler Sprachmerkmale -- Idealerweise gleich Null.
Auch dieses Ziel ist im Hinblick auf eine vereinfachte Applikationsentwicklung, aber auch eine einfachere Benutzbarkeit durch Menschen auf dem Wege der Komplexitätsreduktion zu interpretieren.
6. Lesbarkeit.
Das entstehende Textformat soll für Menschen und Maschinen gleichermaßen les- und verstehbar sein.
7. Kompakte Spezifikation.
Die entstehende XML-Spezifikation sollte deutlich weniger Umfang aufweisen als der SGML-Vorgängerstandard. Letztlich konnte die reine Seitenzahl von über 600 Seiten für die SGML-Spezifikation auf ungefähr 30 Seiten für XML reduziert werden.
8. Formaler und präziser Sprachentwurf.
Um die schnelle Akzeptanz seitens der Anwender zu forcieren erachteten die Mitglieder der XML-Arbeitsgruppe die schnelle Verfügbarkeit von XML-Werkzeugen für essentiell. Aus diesem Grunde sollte der XML-Sprachentwurf möglichst leicht und eindeutig in XML-Werkzeuge zu implementieren sein.
9. Leichte Dokumenterstellung.
Die Erstellung von korrekten XML-Dokumenten sollte idealerweise so einfach sein, daß hierfür keine speziellen Werkzeuge benötigt werden.
10. Nicht notwendigerweise knappes Markup.
Kompaktheit und Effizienz hinsichtlich des Volumens eines XML-Dokuments war zu keinem Zeitpunkt eines der Hauptentwicklungsziele. Auf der Basis des XML-Information Sets ist es jedoch möglich beliebig kompakte Binärformate identischer Mächtigkeit zur die in der XML-Spezifikation vorgestellten Textnotation zu definieren.

XML stellt jedoch keine echte semantische Auszeichnungssprache dar, da durch die Metasprache lediglich eine Möglichkeit zur Formulierung eigener Syntax gegeben ist. Die Bedeutung der Elemente bleibt jedoch unberücksichtigt, und kann mittels XML nicht ausgedrückt werden.

Tabelle 2: Einige chronologische Eckdaten



Jahr	Ereignis
1945	Vannevar Bush diskutiert in seinem Artikel As We May Think ein persönliches Informationssystem mit Kommunikationsmöglichkeiten und Zugriff auf Bücher, Tonaufnahmen, etc. unter dem Namen <i>Memex</i> .
1967	William Tunnicliffe (Chairman des Graphic Communications Association (GCA) Composition Committee) schlägt aus seinen Erfahrungen bei der wiederholten Erstellung von Telefonkatalogen (<i>yellow pages</i>) vor, häufig auftretende strukturelle Elemente zu standardisieren.
September 1967	William Tunnicliffe (Vorsitzender der Graphic Communication Association) spricht sich auf einer Konferenz des <i>Printing Office</i> der Regierung von Kanada für die Separierung von Inhalt und Format aus.
Ende der 1960er Jahre	Stanley Rice, ein New Yorker Schriftsetzer, schlägt <i>editorial structure tags</i> vor. Der CGA-Direktor Norman Scharpf initiiert das Projekt <i>GenCode</i> .
1969	Charles Goldfarb, Edward Mosher und Raymond Lorie entwickeln bei der IBM die <i>Generalized Markup Language</i> (GML). Anwendungshintergrund war ein Projekt zur Integration von Informationssystemen für Anwaltskanzleien.
1970	Goldfarb formuliert zwei Grundprinzipien generalisierter Auszeichnungssprachen: 1) Auszeichnungssprachen beschreiben die Dokumentstruktur, nicht die physischen Charakteristika wie Präsentation 2) Die Struktur der Auszeichnungssprache soll so gewählt sein, daß sie sowohl von Menschen als auch Maschinen interpretiert werden kann
1978	ANSI ruft <i>Computer Languages for the Processing of Text</i> -Komitee ins Leben. Ziel ist die Weiterentwicklung der GML zu einem nationalen US-Standard.
1980	• ANSI veröffentlicht ersten Entwurf einer standardisierten GML (SGML). • Tim Berners-Lee tritt seine Arbeit am Europäischen Kernforschungszentrum CERN an. Dort entwickelt er in der Folgezeit die (niemals veröffentlichte) Hypertextanwendung <i>Enquire</i>.
1983	Der International Revenue Service (IRS) und das US Verteidigungsministerium (DoD) übernehmen den sechsten Entwurf zur SGML (auch bekannt als GCA 101-1983).
1984	Die SGML-Arbeitsgruppe nimmt unter Schirmherrschaft der International Standardization Organization (ISO) als ISO/IEC JTC1/SC18/WG8 ihre Arbeit auf. Goldfarb dient als <i>technical leader</i> der ISO-Gruppe, sowie dem umorganisierten ANSI-Komitee X3V1.8.
1985	Norm-Entwurf zu SGML veröffentlicht.
15. Oktober 1986	ISO verabschiedet SGML als ISO 8879:1986.
März 1989	Berners-Lee schlägt mit dem Dokument Information Management: A Proposal ein SGML-basiertes Hypertext-System zum Informationsaustausch vor.
1990	Am Weihnachtstag nimmt das World Wide Web seinen Betrieb mit zwei Maschinen am CERN auf. Die notwendigen Implementierungen von HTML, HTTP und URL erfolgten durch Berners-Lee . Die erste WWW-Verbindung wird zwischen Berners-Lees Workstation und Robert Cailliaus' NeXT-Rechner aufgebaut. Ein Screenshot des ersten Web-Browsers NeXTStep-Implementierung des Browsers
1991	Beginn der turnusmäßigen Überarbeitungsphase von ISO 8879.
3. November 1992	Erster Entwurf zu HTML
Juni 1993	Einreichung des ersten HTML Entwurfs bei IETF .
Oktober 1994	Gründung World Wide Web Consortium
14. November 1996	Erster Entwurf zu XML vorgestellt
14. Januar 1997	Verabschiedung der HTML v3.2
1998	W3C gibt die erste Version von XML als Recommendation frei.

2000	• W3C gibt XHTML v1.0 -- die Reformulierung von HTML v4.01 zu einer XML-Anwendung -- frei. • W3C verabschiedet XML 2nd edition; sie integriert u.a. die XML Namespaces und behebt einige editorielle Fehler.
2. Mai 2001	Das W3C verabschiedet den XML Schema -Standard. Er geht an vielen Stellen deutlich über die ererbten SGML-Möglichkeiten hinaus, und markiert den Übergang von Präsentations-orientierten Strukturen hin zu Datenstrukturen.

Zum Abschluß dieser Einführung seien die zehn Punkte zusammengestellt und kommentiert, die durch das World Wide Web Consortium als plakative Kurzcharakterisierung von XML [veröffentlicht](#) wurden:

1. XML steht für strukturierte Daten.

Diese Aussage betont die Rolle von XML als Sprache um Sprachen zu erzeugen. Nicht XML wird innerhalb verschiedenster Applikationen direkt verarbeitet, sondern XML-basierte Formate. So steht nicht die XML selbst für all diese Anwendungsdomänen, sondern die jeweiligen problemspezifischen XML-basierten Sprachen. XML selbst dient lediglich der Strukturierung der verschiedensten darzustellenden Daten.

Gleichzeitig rückt durch Aussage die Rolle der XML als Datenformat in den Vordergrund und läßt so die Weiterentwicklung gegenüber den präsentationsorientierten Vorläufern deutlich werden. Die Vorlesungskapitel [Strukturelle Grundkonzepte](#) und [XML Schemasprachen](#) vermitteln einen Eindruck dieses Wandels und dokumentieren die Grundlagen des gegenwärtigen datenorientierten Einsatzes der XML.

2. XML sieht ein wenig wie HTML aus.

Diese Aussage soll offenkundig einerseits den bisherigen HTML-verwendenden Web-Autoren den Einstieg in die XML schmackhaft werden lassen. Dennoch führt sie ein wenig von der Grundidee XMLs als generischer Auszeichnungssprache für beliebigste Anwendungen weg, indem sie den Blick auf HTML focussiert.

Die -- im Grunde der Verwandtschaft zu SGML geschuldete -- offensichtliche syntaktische Ähnlichkeit zu HTML wird bereits bei der Betrachtung der [strukturellen Grundkonzepte](#) deutlich.

3. XML ist Text, aber nicht zum Lesen.

XML-Dokumente können sicherlich im wörtlichen Sinne „gelesen“ werden ... Die Aussage zielt jedoch auf den intendierten Einsatzzweck von XML: der Darstellung von Daten für den Austausch zwischen Maschinen. Unbenommen dessen kann XML selbstverständlich auch von Menschen gelesen und verstanden werden, wenngleich dies bei umfangreicheren XML-Dokumenten durchaus mühsam werden kann.

Aufschluß über die textuelle Natur XMLs, insbesondere im Hinblick auf die Verwendung unterschiedlicher Alphabete, liefert das Kapitel [strukturelle Grundkonzepte](#).

4. XML ist vom Design her ausführlich.

Hiermit wird versucht dem häufig geäußerten Kritikpunkt der Platzzunahme XML-codierter Inhalte gegenüber klassischen Darstellungsweisen etwas pauschal entkräftend entgegenzutreten. Sicherlich geht das W3C in dieser Aussage nicht fehl, wenn die Entwicklung der Netzwerkbandbreiten, der CPU-Leistung und der Speicherkapazitäten berücksichtigt. Andererseits ist die Aufblähung der XML-formatierten Inhalte im Vergleich zu optimierten Binärformaten nicht von der Hand zu weisen, wird jedoch durch die mit der Verwendung von XML einhergehenden Vorteile mehr als ausgeglichen.

Einen ersten Eindruck der Natur XML-codierter Inhalte liefert das Kapitel [strukturelle Grundkonzepte](#). Dort finden sich auch Ansätze die bekannte XML-Syntax kompaktifiziert darzustellen ohne die Vorteile der generischen Auszeichnungssprache aufgeben zu müssen.

5. XML ist eine Familie von Techniken.

Eine Aussage durch alle drei Kapitel der Vorlesung unterstrichen wird, die deutlich zeigen, daß XML nicht als isolierte Idee oder Technik anzusehen ist -- sondern erst im Zusammenspiel mit anderen XML-Standards und eingebettet in Applikationen und Infrastrukturen -- seine volle Wirkungsmächtigkeit entfalten kann.

6. XML ist neu, aber nicht so neu.

Diese Bezugnahme soll nochmals unterstreichen, daß XML keineswegs den Anspruch erhebt eine vollkommen neue technische Errungenschaft zu sein, sondern vielfach bekanntes und erprobtes aus der Informatik wiederverwendet und im neuen Verwendungskontext weiterentwickelt. Diese Aussage wird durch die in den einzelnen Kapiteln dargebotenen Rückbezüge auf bereits bekannte Techniken und Lösungsformen untermauert.

7. XML überführt HTML in XHTML.

Diese Aussage greift nochmals die Beziehung zwischen XML und HTML auf. Diesmal soll die Rolle von XML im Bezug auf die Weiterentwicklung von HTML zum XML-basierten Vokabular *XHTML* unterstrichen werden. So löst XML die Abhängigkeit zwischen SGML und HTML auf und

reformuliert HTML auf der Basis von XML.

Das Kapitel [XHTML](#) führt kurz in die Entwicklung der neuen HTML-Varianten auf Basis der XML ein und skizziert die vorgenommenen Änderungen und zukünftige Erweiterungen dieser Hypertextsprache.

8. XML ist modular.

Hierdurch wird unterstrichen, daß XML kein in sich geschlossenes monolithisches Gebilde darstellt, sondern einzelne Vertreter aus der Familie der XML-Sprachen wahlfrei zur Lösung konkreter Probleme herangezogen werden können. Ebenso wird die Sprachfamilie beständig an verschiedensten Stellen unabhängig voneinander weiterentwickelt, ohne einer zentralen Koordination zu bedürfen.

9. XML ist die Basis für RDF und das Semantic Web.

Grundidee des Semantic Web ist die Weiterentwicklung des sichtbaren XHTML-basierten Webs unter Nutzung seiner datenorientierten Ergänzung XML zu einem Netz von Sinnzusammenhängen.

10. XML ist lizenzfrei, plattform- und herstellerunabhängig, und gut unterstützt.

XML ist eine durch das [World Wide Web Consortium](#) herausgegebene Spezifikation, die kostenfrei über das Web bezogen werden kann und durch Interessierte ohne weitere Lizenzkosten in eigenen kommerziellen Produkten verwendet werden. Durch den Standardisierungsprozeß innerhalb des World Wide Web Consortiums wird sichergestellt, daß keine Ausführungsplattform bevorzugt wird und gleichzeitig keine Nachteile für Andere entstehen. Dies wird durch die herstellerunabhängige Organisation des Gremiums versucht zu garantieren, in dem zwar Hersteller Mitglied werden können, die technischen Entscheidungen jedoch Arbeitsgruppen obliegen, die nicht durch eine Firma dominiert werden können.

Web-Referenzen 1: Vertiefende Informationen



- Artikel in der Online-Ausgabe des *Economist* über Ted Nelson -- [The Babbage of the web](#)
- [COT1800 Public Networks, Lecture 8, Standard Generalised Markup Language](#)
- [Brief History of Document Markup](#)
- [XML, Element Types, DTDs, and All That](#)
- [Clark, J.: Comparison of SGML and XML](#)

Definition 2: XML-Sprache



Eine Anwendung der Extensible Markup Language. Ein Vokabular, das aus Symbolen und der ihnen zugewiesenen Bedeutung (Semantik) gebildet wird, ergänzt um Regeln (grammatikalische Struktur und Gültigkeitsregeln für den Inhalt (z.B. Datentypen)) zur Kombination der Vokabularelemente.

Anwendungen einer so neu geschaffenen XML-Sprache *L* werden als XML-Dokumente, auch: *L*-Dokumente, bezeichnet.

Strukturelle Grundkonzepte

Die grundlegende XML-Syntax ist in der namensgebenden W3C-Recommendation der [Extensible Markup Language](#) definiert. Die Semantik der Metasprache wird hingegen durch den W3C-Standard des [XML Information Set](#) festgelegt.

Diese Spezifikationen beinhalten die grundlegenden Definitionen hinsichtlich Terminologie und Beziehung der verschiedenen möglichen Elemente eines XML-Dokuments. Im vorliegenden Teilkapitel werden beide Sprachaspekte grundlegend eingeführt und ein erstes Verständnis der XML vermittelt. Dabei wird in Form von Ausblicken auf nachfolgende Abschnitte der Bogen zu Grammatikdefinitionssprachen und weiterführenden Konzepten wie Namensräumen gespannt. Zum leichteren Verständnis sind die aus der offiziellen Spezifikationen entnommenen formalen Grammatikdefinitionen der [EBNF](#)-Notation durch vereinfachte graphische Strukturdarstellungen ergänzt.



Definition 3: XML Dokument

Ein XML-Dokument ist ein Datenstrom (der nicht zwingend als Datei vorliegen muß), welcher den [Strukturierungsprinzipien](#) der eXtensible Markup Language genügt.

Definition 4: XML Information Set



Die Spezifikation des XML Information Sets definiert die Semantik der Metasprache XML, d.h. ihre zentralen Begriffe.

Gleichzeitig setzt es diese Begriffe in Beziehung und definiert so syntaxunabhängig die Struktur eines XML-Dokumentes.

Ausgehend von der Allgemeinheit der Aussage aus Definition 1 folgt, daß der Infoset neben seinem theoretischen Wert als Semantikdefinition zur XML auch zur Formulierung der Datenstrukturen, welche innerhalb eines XML-Prozessors vorliegen müssen, um beliebige XML-Dokumente

verarbeiten zu können, herangezogen werden kann.

Daher läßt sich ein XML-Prozessor definieren als:

Definition 5: XML-Prozessor



Ein XML-Prozessor ist eine maschinelle Komponente (typischerweise: Software), die zum Lesen, Speichern und Verarbeiten eines XML-Dokuments eingesetzt wird. Er erlaubt Zugriff auf den Inhalt und die Struktur des XML-Dokuments.

Die XML-Spezifikation faßt den XML-Prozessorbegriff etwas enger und beschränkt ihn lediglich auf Software-Module, die XML-Dokumente lesend verarbeiten. Konzeptionell spricht jedoch nichts gegen eine Umsetzung in Hardware, beispielsweise im Kontext eingebetteter Systeme etc. ([In XML-Spezifikation nachschlagen](#))

Ferner nimmt die XML-Spezifikation an, ein Prozessor operiere nicht eigenständig, sondern im integrierten Zusammenspiel mit einer Applikation.

Beispiel 1: Ein erstes XML-Dokument

```
(1) <?xml version="1.0" encoding="ISO-8859-15" standalone="yes"?>
(2) <Vorlesung>
(3)   <Pflichtfach/>
(4)   SS2003
(5)   <Titel beginn="2003-03-17T14:00:00+01:00">eBusiness-Engineering</Titel>
(6)   <Hochschule>Fachhochschule Furtwangen</Hochschule>
(7)   <Praktikum>Kein Übungsbetrieb</Praktikum>
(8) </Vorlesung>
```



Download des Beispiels

Das Beispiel zeigt ein erstes einfaches XML-Dokument, welches bereits die häufigst verwendeten Sprachelemente der XML versammelt.

Jedem XML-Dokument entspricht genau ein Information Set, der alle Informationselemente des Dokuments in Form einer Baumstruktur beinhaltet. Die nachfolgende Abbildung zeigt den Information Set des Beispiels in der Notation eines UML-Klassendiagramms. Dabei sind die einzelnen Knoten des Information Sets als Objekte (Klassensymbole mit unterstrichenem Klassennamen) und die Eigenschaften der Knoten als Attributwerte dargestellt.

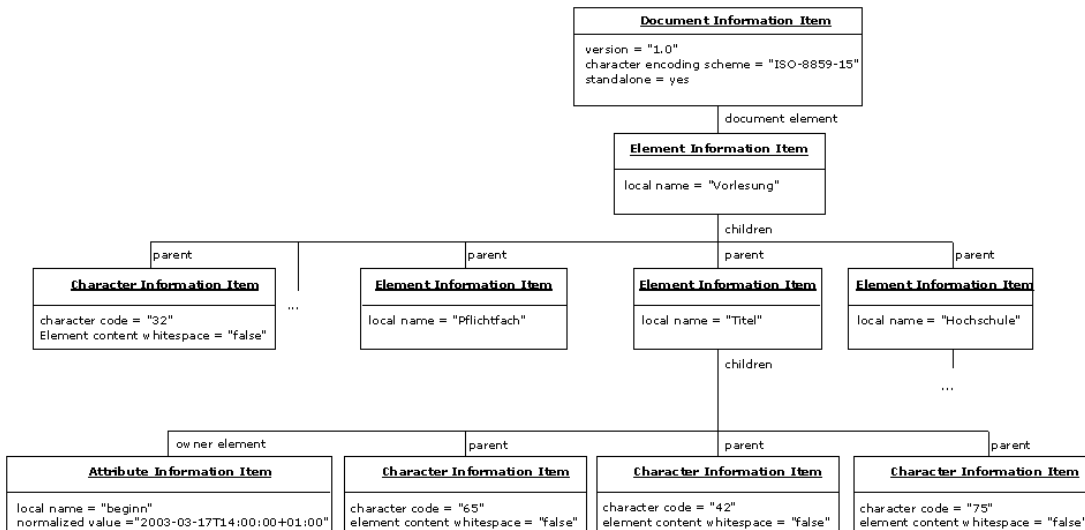


Abbildung 1: Darstellung des Information Sets zu Beispiel 1 als UML-Klassendiagramm ([click on image to enlarge!](#))

Document Information Item

Jedes Information Set besteht genau aus einem *Document Information Item*. Dieses stellt den äußeren Rahmen des XML-Dokuments dar. Es beinhaltet dokumentbezogene Informationen, wie die verwendete XML-Version und das gewählte Codierungsschema innerhalb des Unicode-Systems. Das Document Information Item enthält daher u.a. die Informationen des [XML-Dokumentprologs](#) in der erste Zeile jedes Dokuments. Das durch die öffnende Winkelklammer und ein Fragezeichen eingeleitete Konstrukt ist in der ersten Zeile des Beispiels 1 dargestellt. Innerhalb des Prologs findet sich die Zeichenkette `xml`, sowie die Bezeichner `version` und `encoding`. Beiden ist ein durch doppelte Hochkommata umschlossener Wert nachgestellt, 1.0 für `version`, bzw. ISO-8859-15 für `encoding`. Beendet wird der Prolog wiederum durch ein Fragezeichen und die schließende Winkelklammer.

Wird auf die Angabe des optionalen Prologs im Dokument verzichtet, so sind die daraus ableitbaren Angaben im Document Information Item nicht gesetzt.

Als weitere Eigenschaften verfügt jedes Document Information Item über eine geordnete Liste von Kindknoten. Darin ist genau ein [Element Information Item](#) enthalten, welches den Startknoten des XML-Dokuments verkörpert. Wegen seiner hervorgehobenen Bedeutung als Wurzel des Dokumentbaumes wird dieser Knoten auch als Document Element bezeichnet. Zusätzlich kann die Liste Elemente vom Typ [Processing Instruction Information Item](#) enthalten. Sie dienen der Darstellung von Verarbeitungsanweisungen, die durch den XML-Prozessor interpretiert werden.

Im Kopfbereich vor Document Element platzierte XML-Kommentare werden durch [Comment Information Items](#) innerhalb der children-Liste dargestellt.

Zusammengefaßt enthält das Document Information Item folgende Informationen:

- **Kindknoten:** In der Reihenfolge des Auftretens im Dokument geordnete Liste. Sie enthält mindestens das Document Information Item. Ferner je ein Element des Typs Processing Instruction Information Item für jede Processing Instruction die außerhalb des Wurzelements definiert ist und jeweils ein Comment Information Item zu jedem definierten Kommentar.
- **Document Element:** Ein Element des Typs Element Information Item, das auf den Wurzelknoten des Dokuments verweist.
- **Basis URI:** Lokation, falls bekannt, des XML-Dokuments in Form eines *Uniform Resource Identifiers* (URI) gemäß [IETF RFC 2396](#).
- **Character Encoding Scheme:** Der Name der gewählten Codetabelle aus dem Unicode-Standard.
- **Standalone:** Legt -- als Boole'scher Wert (Zugelassene Belegungen: *yes*, *no*) -- fest, ob die im Dokument gespeicherten Daten durch eine sie verarbeitende Applikation interpretiert und somit ergänzt oder verändert werden.

Häufigste Form dieses Interpretationsvorganges ist die Präsenz einer expliziten Grammatik in Form einer *Document Type Definition*, welche Gültigkeitsregeln für eine Familie von Dokumenten formuliert.

Konsequenterweise bedeutet daher die Belegung mit *no*, daß die gespeicherten Daten entweder durch die Applikation interpretiert werden, dies ist beispielsweise bei der Auflösung von Entitäten der Fall, oder durch eine DOCTYPE-Deklaration ein Verweis auf die externe Dokumentgrammatik erfolgt. Die Angabe von *standalone* ist optional. Fehlt sie und ist gleichzeitig eine DOCTYPE-Deklaration im Dokument gegeben, so wird *standalone="no"* angenommen.

Im [Beispiel](#) ist die standalone-Deklaration auf *yes* gesetzt, d.h. es existiert explizit keine Dokumentgrammatik. ([In XML-Spezifikation nachschlagen](#))

- **Version:** Die eingesetzte XML-Version. Dieser Wert wird aus dem Dokumentprolog übernommen.

Wie auch im Beispieldokument, bildet die erste Zeile den sog. *Prolog* eines jeden XML-Dokuments ([In XML-Spezifikation nachschlagen](#)). Die Angabe der Version ist zwingend und derzeit auf die Konstante 1.0 fixiert. Die aktuelle XML-Spezifikation sieht als gültige Belegung der Versionsangabe ausschließlich die Zeichenkette 1.0 vor. Zukünftigen Weiterentwicklungen ist es jedoch freigestellt auch andere Revisionskennungen zu vergeben.

encoding leitet das zweite Namen-Wert-Paar ein. Die Deklaration ist innerhalb des Prologs optional, und kann daher auch unterbleiben. Die Zeichenkette der Encodingdeklaration benennt das Codierungsschema, welches für das so gekennzeichnete Dokument verwendet wurde. Es definiert den Satz der innerhalb des Dokumentes zugelassenen Zeichen fest.

Gemäß [Produktion 22 der XML-Syntaxdefinition](#) ist der gesamte Prolog optional.

Die Encoding-Deklaration hat folgendes Aussehen ([In XML-Spezifikation nachschlagen](#)) :

```
[80] EncodingDecl ::= S 'encoding' Eq ( '"' EncName '"' | "'" EncName "'" )
[81] EncName      ::= [A-Za-z] ([A-Za-z0-9._] | '-' ) *
[3] S            ::= (#x20 | #x9 | #xD | #xA) +
[25] Eq          ::= S? '=' S?
```

Die Festlegung der Produktion 80, sowie die der [Produktion 23](#), stellt heraus, daß sich die Encodingdeklaration nicht auf die Prologzeile selbst auswirkt. Hier sind die beiden Zeichenketten xml und encoding in der Codierung UTF-8 oder UTF-16 Vorschrift.

Als Belegungen des Encoding Namens (EncName) sind beliebige Zeichensätze zugelassen. Der XML-Standard empfiehlt jedoch lediglich auf die durch die *Internet Assigned Numbers Authority* verwalteten zurückzugreifen ([Dokument: Official Names for Character Sets](#)) ([In XML-Spezifikation nachschlagen](#)).

Die häufigsten praktisch eingesetzten Deklarationen sind die der ISO-8859 (*extended ASCII*)-Familie, sowie die der Unicode- und ISO-10646-Standards.

Die verschiedenen Abschnitte der ISO-8859 Familie werden als ISO-8851-*n* ausgedrückt, wobei *n*

die Nummer des Abschnittes des zugehörigen ISO-Dokuments referenziert. Ferner können die durch JIS X-0208-1997 normierten asiatischen Zeichensätze als ISO-2022-JP, Shift_JIS und EUC-JP dargestellt werden.

```
<?xml version="1.0" encoding="Shift_JIS" standalone="yes"?>
<kougi>
  <title begin="二千一年三月二十一日三時三十分+九時">選挙義務講義XML</title>
  <organization>アウグスブルグ大学コンピュータ・サイエンス過分、工学と家政と形成の大学</organization>
</kougi>
```

Abbildung 2: Das Beispiel als japanisches XML-Dokument (click on image to enlarge!)

Unicode stellt einen Industriestandard (entwickelt u.a. durch Apple, HP, IBM, Microsoft und SUN) zur Darstellung verschiedenster Alphabete und graphischer Zeichen dar. Sein zunächst durch 16-Bit codierter Zeichenvorrat bot Raum für 65536 unterschiedliche Symbole.

Die seit 1991 laufenden Unicodebemühungen mündeten in die ISO-Norm zur Erweiterung des klassischen ASCII-Codes (ISO 646) als ISO-10646 *Universal Multiple-Octet Coded Character Set* (UCS). Seit 1996 sind beide Standards synchronisiert und werden abgestimmt vorangetrieben. UCS definiert zwei aufeinander aufbauende Codierungen: UCS-2 (16 Bit Umfang) und UCS-4 (32 Bit). Der bisherige Unicode-Standard ist voll kompatibel zu UCS-2 und durch diesen darstellbar.

Tabelle 3: Verschiedene Codierungen des Zeichens "A"



Codierung	Bitbreite	Binärdarstellung	Größe der Beispieldatei in Byte (ohne Berücksichtigung des XML-Prologs)	Bemerkung zum Meßwert
UTF-7	>= 7	100 0001	263	(encoding="UTF-7")
Extended ASCII, Latin-1 (ISO-8859-1)	8	0100 0001	258	(encoding="ISO-8859-1")
UTF-8	>= 8	0100 0001	259	(encoding="UTF-8") keine Byte Order Mark
UCS-2, Unicode	16	0000 0000 0100 0001	516	(encoding="UCS-2") keine Byte Order Mark
UTF-16 (big endian)	>= 16	0000 0000 0100 0001	516	(encoding="UTF-16") keine Byte Order Mark
UCS-4	32	0000 0000 0000 0000 0000 0000 0100 0001	1032	(encoding="UTF-8") keine Byte Order Mark
UTF-32	>= 32	0000 0000 0000 0000 0000 0000 0100 0001	1032	(encoding="UTF-32") keine Byte Order Mark

Die Zeilenumbrüche wurden in allen Fällen durch die Kombination von Wagenrücklauf und Zeilenvorschub ausgedrückt.

Die Tabelle stellt einige Codierungen zur Darstellung des Zeichens A zusammen. Auffallend ist der große Platzbedarf der UCS-2 und -4 Codierungen. Insbesondere bei den „klassischen“ ASCII-Symbolen werden hier (u.U. sehr viele) führende Nullbits erzeugt, die in der Konsequenz zu einer deutlichen Vergrößerung der Beispieldatei führen. Daher wurde mit dem *UCS Transformation Format* (UTF) eine kompaktere Darstellung zum jeweiligen UCS-Set eingeführt. UTF-8 verwendet standardmäßig die ersten acht Bit zur Darstellung der bekannten ASCII-Zeichen

Anmerkung: Inzwischen existiert auch eine „UTF-32“ genannte 32-Bit Ausprägung, diese ist jedoch identisch zu UCS-4, mit Ausnahme daß durch UTF-32 „nur“ 2^{21} -Zeichen dargestellt werden können. Die Dateigröße ist daher für das betrachtete Beispiel in dieser Darstellungsweise unverändert zu der des UCS-4-Encodings.

Der Größenunterschied zwischen der UTF-7 codierten Datei und der Latin-1 encodierten erklärt sich aus der Darstellung des Umlautes sowie des +-Zeichens, die beide nicht im klassischen 7-Bit ASCII-Code enthalten ist. So wird Ü im Wort *Übungsbetrieb* des Beispieldokumentes durch die die

Bytefolge 2B 41 4E 77 2D dargestellt, während alle übrigen Zeichen durch ein einzelnes Byte ausgedrückt werden können.

UTF-8 ist in der Lage sämtliche Standard-ASCII-Zeichen durch jeweils genau ein Byte auszudrücken, wiederum für den Umlaut muß auf die 16-Bit-Darstellung des UCS-2 zurückgegriffen werden. Daher erhöht sich hier die Dateigröße um ein Byte.

Erwartungsgemäß beträgt der Umfang des UCS-2 codierten Dokuments exakt das Doppelte des 8-Bit Äquivalents der Latin-1-Darstellung.

Dasselbe gilt für die UTF-16-Variante, die für das vorliegende Beispiel unterschiedslos zu UCS-4 verläuft, da keinerlei Zeichen aus UCS-4 im Dokument auftreten.

Die nachfolgende Tabelle stellt beispielhaft die Anwendung der UTF-8-Codierung zusammen:

Tabelle 4: UTF-8 Codierung

Unicode-Bereich	Bitbelegung
U-00000000 - U-0000007F:	0xxxxxxx
U-00000080 - U-000007FF:	110xxxxx 10xxxxxx
U-00000800 - U-0000FFFF:	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 - U-001FFFFF:	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
U-00200000 - U-03FFFFFF:	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
U-04000000 - U-7FFFFFFF:	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx



Diese Mimik zeigt den Nachteil des UTF-*n*-Encodings deutlich: Die Darstellung nicht *n*-Bit darstellbarer Zeichen benötigt u.U. mehr Bitstellen als im Standard UCS-Code.

So wird beispielsweise das Zeichen mit der größtmöglichen Position (7FFFFFFF) in UTF durch sechs Byte encodiert, während UCS dieselbe Information mit den verfügbaren 32-Bit ausdrücken kann. Andererseits „verschwendet“ die UCS-Darstellung für die niederwertigen Zeichen Bitstellen durch die führenden Nullen.

In der Praxis gilt es daher für das zu wählende Encoding einen möglichst guten Kompromiß zu finden: Im allgemeinen stellt das UTF-8-Encoding einen solchen dar, soweit überwiegend ASCII-Zeichen, und nur vereinzelt Sonderzeichen (hierzu zählen auch die deutschen Umlaute) eingesetzt werden.

Bei überwiegender Verwendung nicht in acht-Bit ASCII darstellbarer Zeichen (z.B. arabischer, chinesischer, etc.) erhöht die dann aufwendigere UTF-8-Codierung die Datenmenge.

So umfaßt die UTF-16-Darstellung des unten abgebildeten Beispieldokuments, welche in diesem Anwendungsfall identisch zu UCS-2 ist, 966 Bytes, während UTF-8 1299 Byte benötigt.

```
<?xml version="1.0" encoding="UTF-8"?>
<شظططططضس س سسسخخىئىئىلؤؤأ>
س سسسخخىئىئىلؤؤأ آظطيقدممقك إلكض غص
أ آظطيقدممقك إلكض غصشظططططضس س
إلكض غصشظططططضس س سسسخخىئىئىلؤؤ
طططضس س سسسخخىئىئىلؤؤ آظطيقدممقك
خىئىئىلؤؤ آظطيقدممقك إلكض غصشظططط
يقدممقك إلكض غصشظططططضس س سسسخ
غصشظطططططضس س سسسخخىئىئىلؤؤ آظط
ضس س سسسخخىئىئىلؤؤ آظطيقدممقك إلكض
ئىئىلؤؤ آظطيقدممقك إلكض غصشظطططط
قددممقك إلكض غصشظططططضس س سسسخخ
غصشظطططططضس س سسسخخىئىئىلؤؤ آظطي
ضس س سسسخخىئىئىلؤؤ آظطيقدممقك إلكض
</شظططططضس س سسسخخىئىئىلؤؤ>
```

Abbildung 3: Ein XML-Dokument mit arabischen Zeichen (click on image to enlarge!)

Achtung: Bereits durch die Unterstützung der beiden ISO-Zeichendarstellungen UTF-8 und UTF-16 ist die Konformität zum XML-Standard erfüllt! XML-Prozessorimplementierungen wird nicht abverlangt darüberhinausgehend weitere Darstellungen umzusetzen. ([In XML-Spezifikation nachschlagen](#))

Wie bereits eingangs angemerkt, erklärt die XML-Spezifikation die Encodingdeklaration sowie den gesamten Prolog-Ausdruck als optionales Element ([In XML-Spezifikation nachschlagen](#)) .

Als Konsequenz geht dabei (auch) die Angabe des gewählten Encodings verloren.

Daher fordert der Anhang F der XML-Spezifikation *Autodetection of Character Encodings* bei einem von UTF-8 oder -16 abweichendem Codierungsschema die zwingende Angabe der XML-Deklaration (`<?xml ...`) ([In XML-Spezifikation nachschlagen](#)) .

Hintergrund dieser Maßnahme ist der Versuch anhand der damit bekannten fünf Zeichen das zugrundeliegende Encoding zu ermitteln.

Diese fünf Zeichen können als stabil angenommen werden, da [Produktion 23](#) und [80](#) diese explizit von einem von UTF-8 oder -16 abweichenden Encoding ausnehmen.

Für Dokumente im deutschen Sprachraum, d.h. XML-Ströme die hauptsächlich aus den um die deutschen Umlaute ergänzten Standard-ASCII-Zeichen bestehen, hat es sich in der Vergangenheit eingebürgert den Zeichensatz latin-1 (ISO-8859-1) zu verwenden, um die Mehrbytedarstellung der Umlaute und weiterer Sonderzeichen in der UTF-Codierung zu umgehen.

Jedoch enthält der latin-1-Zeichensatz nicht das unter Unicode-Zeichennummer 20AC abgelegte Eurosymbol (€) welches zur Abkürzung des Währungsbegriffes der europäischen Gemeinschaftswährung verwendet wird.

Dieses Symbol wurde in die unter Nummer 15 veröffentlichte aktualisierte Fassung der Zeichensatzfamilie 8859 aufgenommen. Daher sollte bei der Erstellung von XML-Dokumenten generell darauf geachtet werden entweder ISO-8859-15 als Codierung zu wählen oder auf die ohnehin ungleich flexiblere UTF-Codierung zurückzugreifen.

Die Darstellung der Abbildung 4 faßt die syntaktischen Elemente abgekürzt zusammen:

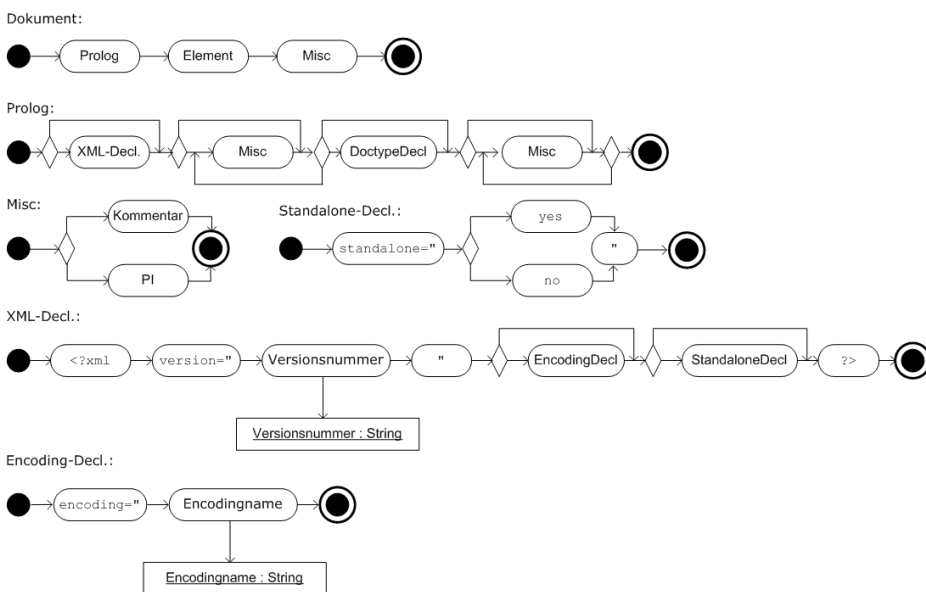


Abbildung 4: Struktur eines XML-Elements (click on image to enlarge!)

Web-Referenzen 2: Weiterführende Links



- Payer, M.: [UNICODE, ISO/IEC 10646, UCS, UTF](#)
- Kuhn, M.: [UTF-8 and Unicode FAQ](#)
- [SC Unipad](#) ein kostenfreier Unicode Editor

Element Information Item

Jedes XML-Dokument enthält mindestens ein *Element*, das Document Element.

Seine, wie auch die Grenzen aller anderen Elemente, werden durch die *Start-* und *Ende-Marke* (engl. *Tag*) markiert. Für den Sonderfall eines *leeren Elements* bildet die Start- auch zugleich die Ende-Marke. Als eine Konsequenz können diese Elemente keine weiteren Kindknoten besitzen.

Die XML-Spezifikation legt den Aufbau des Start-Tags wie folgt fest ([In XML-Spezifikation nachschlagen](#)) :

[40] STag ::= '<' Name (S Attribute)* S? '>'

[41] Attribute ::= Name Eq AttValue

Mittels der Tag-Namen werden die Typen eines Dokumentes definiert. Sie werden später, in Verbindung mit einem Grammatikmechanismus wie [XML-Schema](#), zur Gültigkeitsprüfung herangezogen.

Der Aufbau der Elementnamen ist ähnlich zu den aus den Programmiersprachen bekannten Regeln. Am Beginn muß ein Buchstabe, ein Unterstrich oder der Doppelpunkt stehen. Darauf können nahezu beliebige Zeichen folgen, die über ihre Unicoderepräsentation genau definiert sind.

Leerzeichen und sog. *white spaces* (vgl. [Produktion 3 der XML-Spezifikation](#)) wie Tabulatoren und Zeilenvorschübe sind nicht zugelassen. Desweiteren darf ein Elementname weder Auszeichnungssymbole, wie die öffnenden und schließenden Winkelklammern, enthalten, noch mit der Zeichenkette *XML* beginnen. Die Zeichenfolge *XML* ist -- in allen Schreibweisen -- für die Standardisierung reserviert und wird ausschließlich in W3C-Dokumenten verwendet. Durch den [Namespace Standard](#) (siehe [Abschnitt 1.3](#)) wird dem Doppelpunkt, als Trennsymbol zwischen Namensraumkürzel und Elementnamen, eine besondere semantische Bedeutung zugeschrieben. Daher sollte -- obwohl er spezifikationsgemäß ein erlaubtes Zeichen darstellt -- von seiner Verwendung in Elementnamen abgesehen werden.

Oftmals wird -- insbesondere in der Praxis -- die existierende und notwendige Unterscheidung zwischen *Tag* und *Element* nicht getroffen.

Die Tags oder Marken drücken beschreibende Information über ein Element aus. Der durch den Tag ausgedrückte Elementname liefert somit lediglich deskriptive Information über die Natur des Elements. Hierzu können Worte einer natürlichen Sprache verwendet werden, jedoch auch beliebige andere identifizierende Zeichenketten. Üblicherweise sind jedoch sprechende Tags anzutreffen.

Über den Tag-Namen hinaus kann ein Startelement auch noch *Attribute* enthalten (Vgl. Produktion 41). Diese sind jedoch nicht vom Typ *Element* und werden daher im Abschnitt [Attribute Information Item](#) betrachtet.

Der Aufbau eines Elementnamens wird durch die Produktionen 4ff definiert ([In XML-Spezifikation nachschlagen](#)) :

```
[4] NameChar ::= Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender
[5] Name      ::= (Letter | '_' | ':') (NameChar)*
[6] Names     ::= Name (S Name)*
[7] Nmtoken   ::= (NameChar)+
[8] Nmtokens  ::= Nmtoken (S Nmtoken)*
```

Im [Beispiel](#) sind Vorlesung, Titel und Hochschule („normale“) Elemente, während Pflichtfach ein leeres Element darstellt.

[Die Abbildung](#) zeigt, daß auf der semantischen Ebene des Information Sets die syntaktische Unterscheidung zwischen Elementknoten mit Kindelementen und leeren Elementen des XML-Dokuments keine Berücksichtigung findet.

Eine Sonderstellung unter den Elementen eines Dokuments nimmt der ausgezeichnete Wurzelknoten ein, er wird auch durch das *Document Information Item* referenziert. Unterhalb dieses Knotens spannt sich der Dokumentbaum auf. Hierfür enthält jedes Element Information Item eine geordnete Menge (*children*) weiterer Elementknoten.

Die durch den Elementnamen verwirklichte Typisierung spiegelt sich im Information Set durch das Attribut *local name* wieder.

Darüberhinaus enthält jedes *Element Information Item* durch die Eigenschaft *namespace name* die Identifikation des Namensraumes, in dem dieses Element plziert ist.

Das Namensraumkürzel, welches zur Identifikation eines Elements herangezogen wird, findet sich in der Eigenschaft *prefix*.

Der *local name* entspricht dem -- um Namensraumkürzel und trennenden Doppelpunkt gekürzten -- wiedergegebenen Elementnamen des XML-Dokuments.

Zusätzlich wird jeder Namensraum, der syntaktisch an die Attributdefinition angelehnt ist, in ein Element der ungeordneten Menge *namespace attributes* abgebildet, welche (nochmals) die Namensräume eines Elements beinhaltet.

Beispiel 2: Element mit deklariertem Namensraum



```
(1) ...
(2)   <myNS:aParent xmlns:myNS="example.com">
(3)       <myNS:aElement/>
(4)   </myNS:aParent>
(5) ...
```

Das Beispiel zeigt das leere Element *aElement* innerhalb des Elements *aParent*. Durch das Elternelement wird der Namensraum *example.com* deklariert und dem Kürzel *myNS* zugewiesen. Gemäß den Prinzipien der Namensräume steht der auf dem Elternknoten deklarierte Namensraum auch in allen Kindknoten zur Verfügung. Daher enthält die Eigenschaft *in-scope namespaces* des Elements *aElement* auch die Namensräume der übergeordneten Elemente.

Das resultierende *Element Information Item* des Knotens *aElement* ergibt sich daher als (der Ausschnitt enthält nur die für das Beispiel relevanten Elemente):

```
local name    = aElement
namespace URI = example.com
prefix        = myNS
```

Nähere Ausführungen zur Bedeutung von Namensräumen und ihrer Verwendung finden sich im Abschnitt [Namensräume](#).

Verweise auf die im Dokumentbaum nachfolgenden Knoten eines Elements werden in einer geordneten Liste *children* gesammelt. Ihre Inhalte sind vom Typ [Element Information Item](#), [Character Information Item](#) und [Comment Information Item](#).

Anhand der beiden Informationstypen *Element Information Item* und *Character Information Item* zeigen sich bereits die beiden Strukturierungsformen eines XML-Dokuments. Einerseits die durch die starke Verwendung von Elementen- und Attributen gekennzeichnete strukturierte Darstellung, andererseits die durch „eingestreuten“ Freitext entstehende charakteristische semistrukturierte Variante.

In beiden Fällen werden die textartigen Inhalte durch *Character Information Items* repräsentiert. Das [Beispiel](#) zeigt die verschiedenen Auftretensformen exemplarisch. Der Inhalt der Elemente *title* und *organization* ist rein Zeichenketten-artig; jedoch mischt vor lesung strukturierten Inhalt (in Form der genannten Elemente) und unstrukturierte Information -- repräsentiert durch den Text 2002/03.

Die XML-Spezifikation prägt für Zeichenketten-artige Inhalte, die optional durch *eingestreute* Elemente angereichert werden, den Begriff [mixed Content](#).

children enthält jedoch keine Verweise auf die Attribute eines Elements. Diese sind durch die separate ungeordnete Menge *attributes* repräsentiert. Die Diskussion der als [Attribute Information Item](#) bezeichneten Mengenelemente findet sich im folgenden.

Die in [der Abbildung dargestellte](#) Beziehung *parent* verbindet jedes Element mit seinem übergeordneten. Als Ziele dieser Referenz sind ausschließlich Ausprägungen von [Document Information Item](#) oder [Element Information Item](#) zugelassen.

Diese Festlegung untermauert nochmals die strikte Baumstruktur eines XML-Dokuments. Andernfalls müßte *parent* als Menge definiert werden.

Attribute Information Item

Das [betrachtete Beispiel](#) enthält, neben den Elementen, auch ein XML-Attribut. Syntaktisch werden Attribute innerhalb eines Start-Tags platziert und durch Namen-Wert-Paare ausgedrückt ([In XML-Spezifikation nachschlagen](#)) .

Der Information Set enthält folgende Eigenschaften zu jedem Attribut:

- **namespace name**: Namensraum des Attributs, falls definiert.
- **Lokaler Name**: Der um das eventuell definierte Namensraumkürzel bereinigte Attributname.
- **Präfix**: Namensraumkürzel des Namensraumes, innerhalb dessen das Attribut platziert ist.
- **Normalisierter Wert**: Normalisierter Attributinhalt. Der Normalisierungsvorgang ist in Abschnitt 3.3.3 der XML-Spezifikation beschrieben ([In XML-Spezifikation nachschlagen](#)) . Unter anderem eliminiert er Zeilenumbrüche innerhalb des Attributinhalts und löst Entitätsreferenzen auf.
- **specified**: Boole'scher Wert, der angibt, ob das Attribut im XML-Dokument auftrat oder aufgrund einer Vorgabebelegung durch die DTD erzeugt wurde.
Zur Ermittlung dieser Eigenschaft des Attribute Information Items ist die Definition und Referenzierung einer expliziten Grammatik notwendig.
- **Attributtyp**: Typ des Attributs. Zugelassene Belegungen sind: ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, NMTOKENS, NOTATION, CDATA, und ENUMERATION.
Zur Ermittlung dieser Eigenschaft ist der Zugriff auf die DTD des Dokumentes notwendig. Ist dies nicht möglich, so ist der *attribute type* mit keinem Wert belegt.
- **Referenzen**: Handelt es sich bei dem Attribut um ein Referenzattribut (d.h. es ist als IDREF(S), ENTITY, ENTITIES oder NOTATION typisiert), so enthält diese Eigenschaft eine Verweisliste auf alle Auftreten des Attributwertes.
- **Eigentümerelement**: Bildet die Entsprechung zur [parent-Eigenschaft des Element Information Item](#). Als solches enthält die Eigenschaft einen Verweis auf das Element, welches das Attribut beherbergt.

Im Vergleich zum *Element Information Item* erlaubt das Attribut keine weitere Unterstrukturierung (im XML-Sinne); insbesondere fehlen mengenwertige Eigenschaften zur Aufnahme der dann

notwendigen Verweise. Stattdessen wird der gesamte Inhalt durch die Eigenschaft *normalized value* dargestellt.

Daher dürfen innerhalb von Attributen keine (Meta-)Symbole wie die öffnende Winkelklammer auftreten, die als Starttags (miß-)interpretiert werden könnten ([In XML-Spezifikation nachschlagen](#)) .

Auch die Form des Auftretens von Attributen innerhalb des definierenden Elements unterscheidet sich von der der Subelemente innerhalb eines Elements. Während Kindelemente durch die geordnete Liste *children* dargestellt werden, können Attribute (formalisiert in der ungeordneten Menge *attributes*) in beliebiger Reihenfolge angegeben werden, ohne die Dokumentsemantik zu verändern. Mehr noch, die Listenkonstruktion erlaubt das unterscheidbare mehrfache Auftreten desselben Elements. Diese Mimik ist für allgemeine Mengen, und damit für Attribute, nicht möglich.

Element vs. Attribut

Der Vergleich der Eigenschaften von Element und Attribut zeigt bereits, daß sich nicht weiter strukturierte Elemente auch durch Attribute darstellen ließen. Dies wirft innerhalb der Betrachtung der Syntax eines XML-Dokuments bereits die Frage nach der Organisation, und damit dem Entwurf, eines solchen auf.

Die bestehende XML-Spezifikation bleibt jedoch eine Anwendungs- oder Einsatzempfehlung zu dieser Fragestellung schuldig.

Aufgrund der inhärenten Einschränkungen der Attributprimitive bietet sich ihr Einsatz nur in einigen Sonderfällen an. Beispielsweise zur Darstellung deskriptiver Information über das enthaltende Element, die nicht Bestandteil der im XML-Dokument dargestellten Information ist. Hierbei kann es sich um Informationen höherer Ordnung, sog. Metainformation handeln.

Generell bieten sich Elemente immer dann an, wenn eine weitere Unterstrukturierung des Inhaltes gewünscht oder vielleicht zukünftig notwendig ist. Die Darstellungsform als Attribut würde in diesem Fall eine strukturelle Umorganisation des XML-Vokabulars erfordern, da die Spezifikation keine Unterstrukturierungsmöglichkeit für Attribute vorsieht.

Darüberhinaus gestatten Attribute keine Wiederverwendung in verschiedenen Bedeutungskontexten, da sie syntaktisch an das umgebende Element gebunden sind. Diese Einschränkung wird zwar durch die Einführung des Standards *XML Schema* weitgehend gemildert, jedoch nicht die zuvor genannte Mächtigkeitseinschränkung. Zusätzlich stellen Attribute die einzige Möglichkeit zur Typisierung des Inhaltes dar solange DTDs verwendet werden. Dieser Punkt dürfte jedoch durch den wachsenden Praxiseinsatz der XML Schemata immer mehr an Bedeutung verlieren.

Die Darstellung der Abbildung 5 faßt die syntaktischen Elemente abgekürzt zusammen:

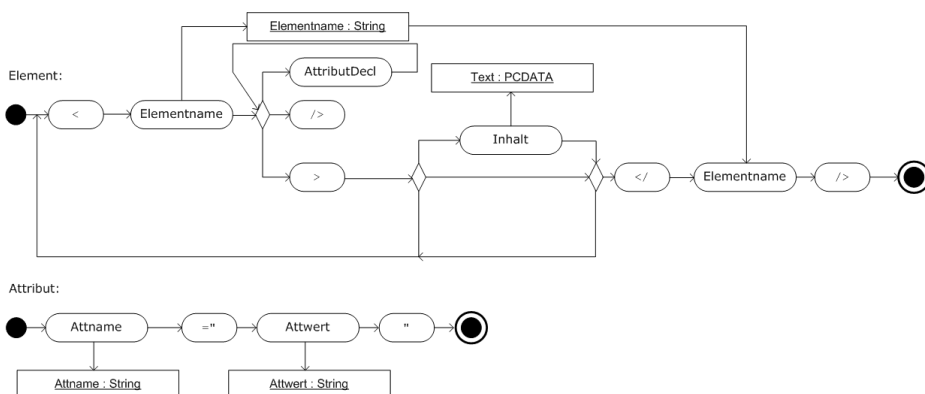


Abbildung 5: Struktur eines XML-Dokuments (click on image to enlarge!)

Character Information Item

Die Betrachtung der Attribut- und Elementknotentypen im Information Set zeigt bereits die zwei grundlegenden Arten der Informationsdarstellung eines XML-Dokumentbaumes.

Die Eigenschaft *normalized value* des *Attribute Information Items* kapselt den im XML-Dokument angegebenen Inhalt direkt im Informationsknoten. Der Datentyp der Eigenschaft ist für alle Dokumenttypen fixiert angebar, da keine weitere Unterstrukturierung von Attributen erfolgen kann. Entgegensetzt hierzu verläuft die Argumentationslinie für Elemente. Ihr Inhaltsmodell kann eine freie Mischung aus Zeichenketten-Daten und weiteren Elementen aufweisen. Die Länge der Zeichenketten ist hierbei nicht näher festgelegt. Daher können diese im minimalen Falle nur aus einem einzelnen Zeichen bestehen. ([In XML-Spezifikation nachschlagen](#)) .

Innerhalb des Information Sets eines Dokuments werden alle Zeichen im Rumpf eines Elements als Ausprägungen des *Character Information Items* dargestellt.

Jedes Character Information Item stellt das im Dokument gegebene Zeichen gemäß ISO 10646-Codierung in der Eigenschaft *character code* dar. Die Werte können hierbei jedoch nur in den durch die Spezifikation vorgegebenen Grenzen variieren ([In XML-Spezifikation nachschlagen](#)).

Darüberhinaus genügt bereits die Unterstützung der UTF-8 und -16-Darstellung zur Erfüllung der Spezifikationsanforderungen an konforme Prozessoren.

Häufig werden white-spaces (Leerzeichen, Tabulator, Zeilenvorschub, Wagenrücklauf) zur besseren visuellen Strukturierung des XML-Dokumentes eingesetzt. So enthält das [Beispieldokument](#) jeweils nach der schließenden Marke einen Zeilenvorschub. Unter Datengesichtspunkten handelt es sich hierbei jedoch um keine verwertbare Information. Die Angabe der Berücksichtigung bzw.

Vernachlässigung im XML-Dokument existierender white-spaces kann in der DTD gesetzt werden. Ist keine solche Deklaration gesetzt oder existiert keine explizite Grammatik, so hat die Eigenschaft *element content whitespace* keinen Inhaltswert.

Der als parent-Eigenschaft realisierte Verweis auf das beherbergende Elternelement bildet den Abschluß der Eigenschaften des *Character Information Items*.

Im betrachteten [Beispiel](#) sind unterhalb der Elemente *organization* und *title* *Character Information Element*-Ausprägungen plazierte. Die [Darstellung](#) zeigt diese als Objekte (Unterhalb des *organization*-Knotens wurde aus Übersichtlichkeitsgründen auf die Darstellung verzichtet).

Eine Sonderrolle kommt den Zeichen zu, die auch als Metasymbole der Auszeichnungssprache dienen. Sie dürfen daher nicht in XML-Dokumenten auftreten.

Bei diesen Zeichen handelt es sich um die beiden Winkelklammern, die einfachen und doppelten Anführungszeichen sowie das *Kaufmanns-Und*. Um eine Fehlinterpretation zu vermeiden existieren hierfür vordefinierte Textersetzungsmuster.

Jeder spezifikationskonforme XML-Prozessor berücksichtigt diese Symbole und gibt sie in der korrekten Darstellung an die Applikation weiter; damit sind diese Fluchtsymbole (engl. *escape characters*) aus Applikationssicht vollkommen transparent.

Tabelle 5: Vordefinierte Textersetzungsmuster



Entitätsreferenz	Ausgedrücktes Zeichen
&	&
<	<
>	>
'	'
"	"



Web-Referenzen 3: Weiterführendes ... Die in XHTML v1.0 vordefinierten Entitäten

[Latin-1 Entities](#)
[Special Entities](#)
[Symbole](#)

Comment Information Item

Zur Dokumentation steht innerhalb jedes XML-Dokuments die von SGML ererbte Kommentierungssyntax zur Verfügung.

Die Spezifikation erlaubt die Anbringung von Kommentaren an zwei Stellen im XML-Dokument:

- Nach dem Prolog. ([In XML-Spezifikation nachschlagen](#))
- An jeder beliebigen Stelle des Inhalts, außerhalb von Markup-Symbolen. ([In XML-Spezifikation nachschlagen](#))

Nicht erlaubt sind demnach Kommentare in Tags, d.h. innerhalb geöffneter Winkelklammern. Dergleichen gilt für Kommentare selbst, was geschachtelte Kommentare verbietet.

Produktion 15 der XML-Spezifikation legt die Struktur wie folgt fest:

```
[15] Comment ::= '<!--' ((Char - '-' ) | ('-' (Char - '-')))* '-->'
```

Als Konsequenz sind innerhalb von Kommentaren alle Zeichen, auch Metasprachensymbole, zugelassen. Somit ist das beliebige „auskommentieren“ von Dokumentteilen möglich.

Als zentrale Einschränkung dürfen (aus SGML-Kompatibilitätsgründen) keine zwei aufeinanderfolgenden Trennstriche (*hyphen-minus*, ISO 10646 #x2D) innerhalb eines Kommentars auftreten, da diese fehlerhafterweise als Beginn des Kommentarendes interpretiert würden.

Der gesamte Inhalt eines Kommentars wird als uninterpretierte Zeichenkette in der Eigenschaft *content* des *Comment Information Items* abgelegt.

Zusätzlich verweist jeder Kommentar über die bekannte *parent*-Eigenschaft auf seinen Elternknoten. Wie bereits durch die beiden Einsatzformen angedeutet, kann es sich hierbei ausschließlich um ein *Document Information Item* oder ein *Element Information Item* handeln.

Beispiel 3: Verschiedene Kommentarstrukturen



```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<Root>
(3)    <!-- this is a comment -->
(4)    <ElementA>
(5)        <ElementB>
(6)            <!--
(7)                <ElementC/>
(8)                <ElementD att1="..." />
(9)            -->
(10)        </ElementB>
(11)    </ElementA>
(12)</Root>
```

Das Beispiel zeigt verschiedene Einsätze von Kommentaren. Zunächst eine einzeilige Anmerkung, die nur verschiedene Zeichen versammelt. Im Anschluß einen mehrzeiligen Kommentar, der auch XML-Strukturen beinhaltet. Ein prozessierender Zugriff auf den Kommentarinhalt ist jedoch nicht vorgesehen, und wird durch gängige Parser und APIs zumeist nicht unterstützt.

Processing Instruction Information Item

Im Gegensatz zu den prinzipiell in beliebigem Freitext formulierbaren Kommentaren, die üblicherweise zur Kommunikation mit einem menschlichen Leser des XML-Dokuments dienen, zielt die *Processing Instruction* und das zugehörige Element des Information Sets auf Kommentare, welche einen maschinellen Verarbeiter des XML-Dokuments, den XML-Prozessor, betreffen.

Im Grunde genommen läuft die Anreicherung eines XML-Dokuments mit Verarbeitungsinformation der Idee einer deskriptiven Auszeichnungssprache entgegen ... Jedoch wurde für die XML beschlossen, nicht zuletzt aus Kompatibilitätsgründen zu SGML, dieses Sprachmerkmal beizubehalten. Eine mögliche weitere Erklärung könnte das syntaktische Aussehen der XML-Deklaration innerhalb des des Dokumentprologs sein. Ihre in [Produktion 23ff](#) festgelegte Struktur stellt eine Anwendung der Processing Instruction dar, auch wenn dies innerhalb der Spezifikation nicht explizit formuliert wird.

Die Syntax einer *Processing Instruction* lautet:

```
[16] PI          ::= '<?' PITarget (S (Char* - (Char* '?'> Char*))?) '?' '
[17] PITarget ::= Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))
```

Eine Processing Instruction wird demnach immer durch eine öffnende Winkelklammer und ein folgendes Fragezeichen eingeleitet. Daran schließt sich die Benennung der Applikation an, für die diese Instruktion eingefügt wurde. Optional können weitere Zeichen -- ausgenommen der Kombination aus Fragezeichen und schließender Winkelklammer -- folgen. Das adressierte System kann beliebig identifiziert werden, jedoch ist die Zeichenkette *XML* in allen Variationen ausgeschlossen. Unbedachterweise verbietet die Spezifikation jedoch nicht die Bildung von Namen, die *XML* als Präfix nutzen ... Jedoch sollte von der Nutzung solcher Konstruktionen abgesehen werden, da sie zur Verwirrung der (menschlichen) Leser beitragen.

Wie Kommentare auch können Processing Instructions an beliebiger Stelle innerhalb des XML-Dokuments auftreten: Vor Beginn des Wurzelements sowie im Rumpf jedes Elements. Nicht gestattet ist ihre Angabe in Elementnamen und Attributen. Ergänzend sei angemerkt, daß die Angabe von Processing Instructions auch innerhalb der *Document Type Definition* erfolgen kann. ([siehe Document Type Definition Information Item](#)).

Beispiel 4: Verschiedene Processing Instructions



```

(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<?mySystem value="42"?>
(3)<root>
(4)  <?System2?>
(5)  <elementA>
(6)    <?System3 a="1" anotherValue?>
(7)  </elementA>
(8)</root>

```

[Download des Beispiels](#)



Übung 1: Processing Instructions

Begründen Sie mit Hilfe der [XML-Spezifikation](#) warum Processing Instructions nicht innerhalb von Elementen und Attributen zugelassen sind.
 Hinweis: Es gibt mehr als eine Begründung!

Das *Processing Instruction Information Item* enthält die angesprochene Zielapplikation als Namen innerhalb der Eigenschaft *target*.

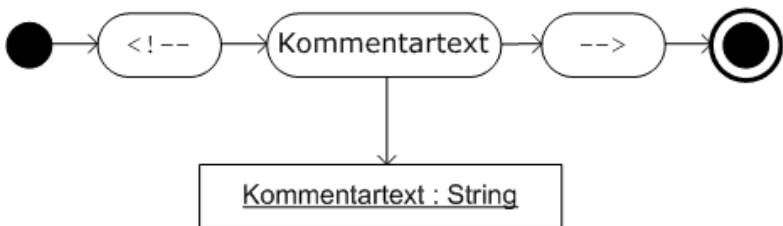
Der weitere Inhalt der Deklaration wird uninterpretiert als Zeichenkette in die Eigenschaft *content* übernommen.

Neben einem Verweis auf die Basis-URI der Processing Instruction wird durch *parent* das Elternelement -- entweder ein Knoten des Typs *Document Information Item* oder *Element Information Item* -- referenziert.

Zur Formalisierung der Identifikation der Zielapplikation empfiehlt die XML-Spezifikation die Verwendung des Sprachmittels *Notation*.

Die Darstellung der Abbildung 6 faßt die syntaktischen Elemente abgekürzt zusammen:

Kommentar:



Processing Instruction:

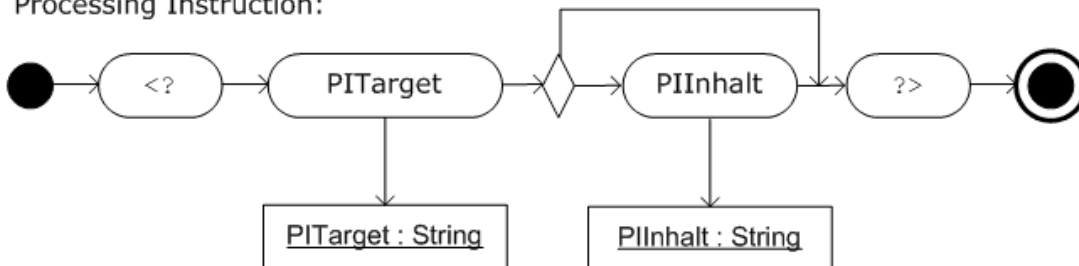


Abbildung 6: Kommentar- und PI-Struktur (click on image to enlarge!)

Namespace Deklaration Information Item

Jedem im XML-Dokument definierten Namensraum ist ein *Namespace Deklaration Information Item* zugeordnet. Es enthält die notwendigen syntaktischen Details zur Identifikation des Namensraumes:

- **prefix:** Das gewählte Präfix des Namensraumes, bzw. leer falls es sich um den Vorgenamensraum handelt.
- **namespace name:** Der Name des Namensraumes, an den das Präfix gebunden ist.

Beispiel 5: Beispiel eines Dokuments mit Namensräumen



```

(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<root>
(3)   <elementA>...</elementA>
(4)   <elementB xmlns="http://www.fh-furtwangen.de">...</elementB>
(5)   <elementC xmlns:abc="http://www.xyz.com">
(6)       ...
(7)       <abc:elementD/>
(8)   </elementC>
(9)</root>

```

Für das Beispiel lauten die Namensräume wie folgt:

Elementname	Namensraum
root	(Das Element befindet sich im leeren Namensraum)
elementA	(Das Element befindet sich im leeren Namensraum)
elementB	http://www.fh-furtwangen.de
elementC	(Das Element befindet sich im leeren Namensraum)
elementD	http://www.xyz.com

Eine ausführliche Betrachtung zur Verwendung von Namensräumen findet sich im entsprechenden [Abschnitt](#).

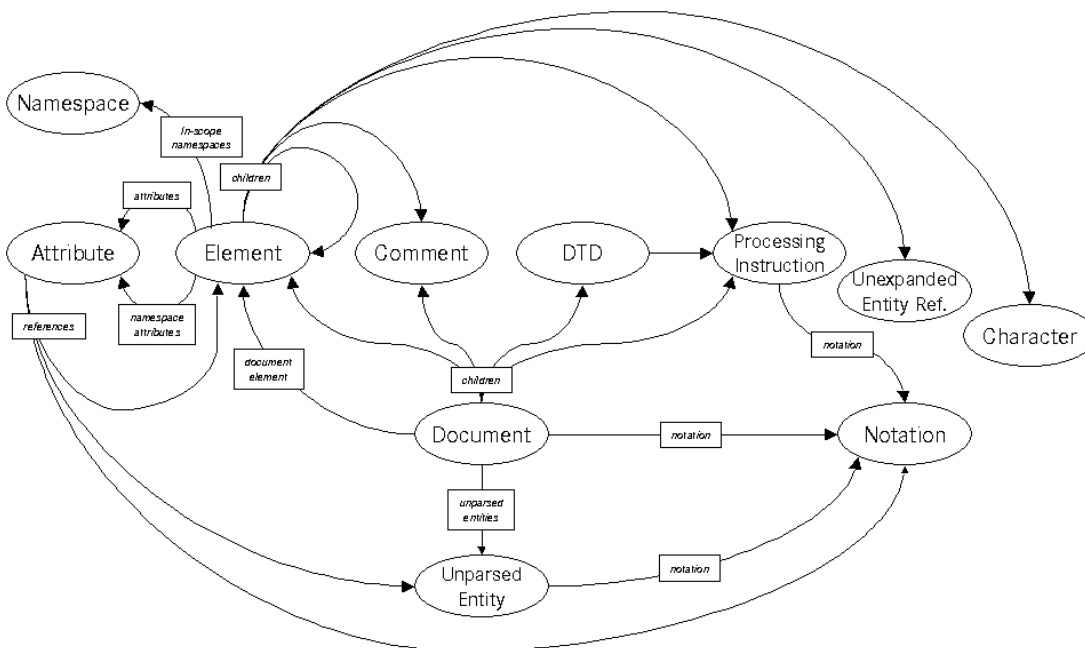


Abbildung 7: Die Elemente des Information Set in der Zusammenstellung *(click on image to enlarge!)*

Die Graphik der Abbildung 7 stellt alle diskutierten Elemente des Information Sets in der Übersicht mit ihren Beziehungen dar. Zur Veranschaulichung wurde eine einfache Graphenstruktur gewählt, die alle Informationseinheiten als Knoten (darstellt als Ellipsen) und alle zugelassenen Beziehungen als gerichtete Kanten zwischen diesen enthält. Zusätzlich ist an die Kanten die Art der Beziehung angetragen.

Den Ausgangspunkt der baumartigen Struktur eines XML-Dokuments bildet die im Zentrum abgebildete Primitive Document Information Item, die alle weiteren Inhalte eines Dokuments über die children-Kante als Kindknoten enthält. Ferner fällt in dieser Darstellung besonders auf, daß lediglich Element Information Items über weitere Kindknoten verfügen und so die charakteristische XML-Struktur herausbilden. Alle übrigen Primitive dienen überwiegend als Blattknoten des Baumes.

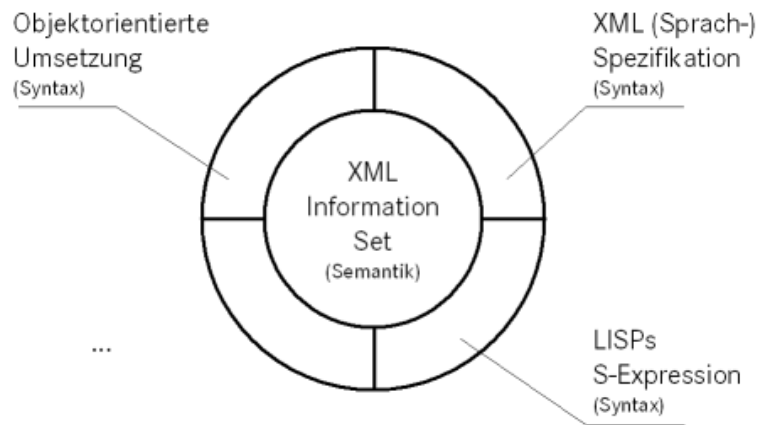


Abbildung 8: Beziehung zwischen XML-Syntax und Semantik (click on image to enlarge!)

Die Graphik der Abbildung 8 setzt die durch den Infoset-Standard definierte Semantik und die darauf aufsetzenden Syntaxen in Beziehung. Der XML-Basisstandard definiert hierbei nur eine von mehreren möglichen Syntaxen zur Darstellung von Infoset-Ausprägungen. Ebenso denkbar wäre der Einsatz anderer Darstellungen gleicher Mächtigkeit wie beispielsweise der S-Expression aus LISP oder objektorientierte Umsetzungen.

Auf Basis der Definitionen des Information Sets lässt sich ein beliebiges XML-Dokument, welches den Strukturierungsprinzipien des Infosets folgt, als *wohlgeformt* (*well-formed*) charakterisieren.

Definition 6: Wohlgeformtes XML-Dokument

Ein textartiges Objekt, dessen Inhalt folgenden Anforderungen genügt:

- Das XML-Dokument nutzt eine DTD, oder enthält die Deklaration `standalone="yes"`
- Zu jedem Start-Tag existiert genau ein Ende-Tag. Bei leeren Elementen können diese zu einem Tag zusammenfallen.
- Korrekte Elementschachtelung, d.h. Elemente überlappen einander nicht.
- Genau ein Wurzelement.
- Alle Attributwerte sind in einfachen oder doppelten Anführungszeichen.
- Kein Start-Tag (oder Tag der ein leeres Element einleitet) enthält zwei oder mehr Attribute desselben Namens.
- Keine Kommentare oder Processing Instructions innerhalb von Tags.
- Kommentare beginnen und enden mit genau zwei Bindestrichen.
- Die Sonderzeichen `<` und `&` treten nicht innerhalb von Elementinhalten oder Attributwerten auf.



[siehe XML-Spezifikation](#)

Der Textstrom des Beispiels 6 zeigt ein nicht-wohlgeformtes XML-Dokument, welches gegen eine Reihe der in Definition 6 verstößt:

Beispiel 6: Ein nicht wohl-geformtes XML-Dokument

```
(1)<?xml version="1.0"?>
(2)<root>
(3)    <elementA att=a oder b>
(4)        <elementB> iff a<b ==> ...
(5)    </elementA>
(6)    <elementC att1="42" att1="3.14">
(7)        <elementD <?do-something?> >
(8)    </elementC>
(9)    </elementD>
(10)    <!-- dies ist nicht erlaubt ---->
(11)</root>
```



[Download des Beispiels](#)

So findet sich in Zeile 3 ein nicht in die erforderlichen Anführungszeichen eingeschlossener Attributwert.

Der textuelle Elementinhalt des in Zeile 4 geöffneten Elements `elementB` enthält ein öffnendes Winkelklammersymbol, welches um Fehler während des Einlesevorganges zu vermeiden durch die alternative Zeichensequenz `<` hätte ersetzt werden müssen. Darüberhinaus fehlt das korrekte schließende Tag zum Öffnenden.

Innerhalb des Elements `elementC` der Zeile 6 wird zweifach ein identisch benanntes Attribut definiert.

Im öffnenden Tag des in Zeile 7 definierten Elements `elementD` findet sich eine `--` dort nicht zugelassene `--` Processing Instruction.

Überdies überlappen sich die Elementgrenzen der Elemente `elementC` und `elementD` und zusätzlich wird der in Zeile 10 platzierte Kommentar nicht durch die erforderlichen genau zwei Bindestriche eingegrenzt.

Namensräume

Die XML-Namensräume wurden schon verschiedentlich erwähnt. Sie bilden die wichtigste, und offensichtlichste Weiterentwicklung der [XML v1.0 2nd edition](#) seit ihrer Veröffentlichung.

Trotz ihrer engen Beziehung zum XML-Kernstandard bildet die Recommendation [Namespaces in XML](#) eine eigenständige Spezifikation. Wegen ihrer engen Beziehung zum XML-Standard und der großen praktischen Bedeutung, sowie ihres Einflusses auf die weitere Entwicklung verschiedenster Sekundärstandards und XML-Sprachen, werden jedoch die Namensräume explizit in der Neuauflage des XML-Standards berücksichtigt. Einen Beleg hierfür bildet die [Anmerkung zu Abschnitt 2.3 Common Syntactic Constructs](#). Dort wird von der `--` laut [Syntaxproduktion 5](#) erlaubten `--` Verwendung des Doppelpunktes in Elementnamen abgeraten. Dies geschieht, um Mehrdeutigkeiten, oder schlichtweg der Verwirrung des Anwenders, vorzubeugen, da es sich beim Doppelpunkt um ein Symbol besonderer Bedeutung innerhalb der Namensraumdeklarationen handelt.

Warum Namensräume?

Die breite Entwicklung immer neuer XML-Sprachen führt zwangsläufig zu Mehrfachentwicklungen für ähnliche oder identische Problemstellungen. Technisch betrachtet äußerst sich dies -- bei natürlichsprachlicher Benennung der Elemente -- durch die Verwendung identischer Bezeichner in verschiedenen XML-Sprachen. Hierbei bilden die verschiedenen Sprachen Anwendungskontexte, innerhalb derer die Bezeichner, durch Einbezug der Anwendungssemantik, eindeutig sind; andernfalls kann unterstellt werden, daß bereits durch die Sprachentwicklung andere Benennungskonventionen gewählt worden wären.

In der Konsequenz der Verfügbarkeit verschiedenster XML-Sprachen für beliebige Anwendungsbereiche entsteht der (berechtigte) Wunsch existierende Sprachfragmente in eigene Sprachen zu integrieren, um so zeitraubenden und vielfach fehleranfälligen Mehrfachentwicklungen vorzubeugen. Jedoch tritt bei diesem Integrationsszenario die u. U. kontextabhängige Elementeindeutigkeit zu Tage.

Das Beispiel zeigt zwei Dokumente identischen Informationsumfanges, die lediglich strukturell differieren.

Beispiel 7: Ein Rechnungsdokument

```
(1)<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
(2)<Rechnung>
(3)  <Kunde>
(4)    <KundenNr>4711</KundenNr>
(5)    <Name>Max Mustermann</Name>
(6)    <Anschrift>
(7)      <Straße>Musterplatz 1</Straße>
(8)      <PLZ>12345</PLZ>
(9)      <Ort>Musterstadt</Ort>
(10)    </Anschrift>
(11)  </Kunde>
(12)  <Rechnungsposten>
(13)    ...
(14)  </Rechnungsposten>
(15)</Rechnung>
```



Beispiel 8: Eine alternative Rechnungsstruktur



```

(1)<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
(2)<Rechnung>
(3)    <Rechnungsanschrift>
(4)        <Kunde kundenNr="4711">
(5)            <Name>Max Mustermann</Name>
(6)            <Straße>Musterplatz 1</Straße>
(7)            <PLZ>12345</PLZ>
(8)            <Ort>Musterstadt</Ort>
(9)        </Kunde>
(10)    </Rechnungsanschrift>
(11)    <Lieferanschrift>
(12)    ...
(13) </Lieferanschrift>
(14) <Rechnungsposten>
(15) ...
(16) </Rechnungsposten>
(17)</Rechnung>

```

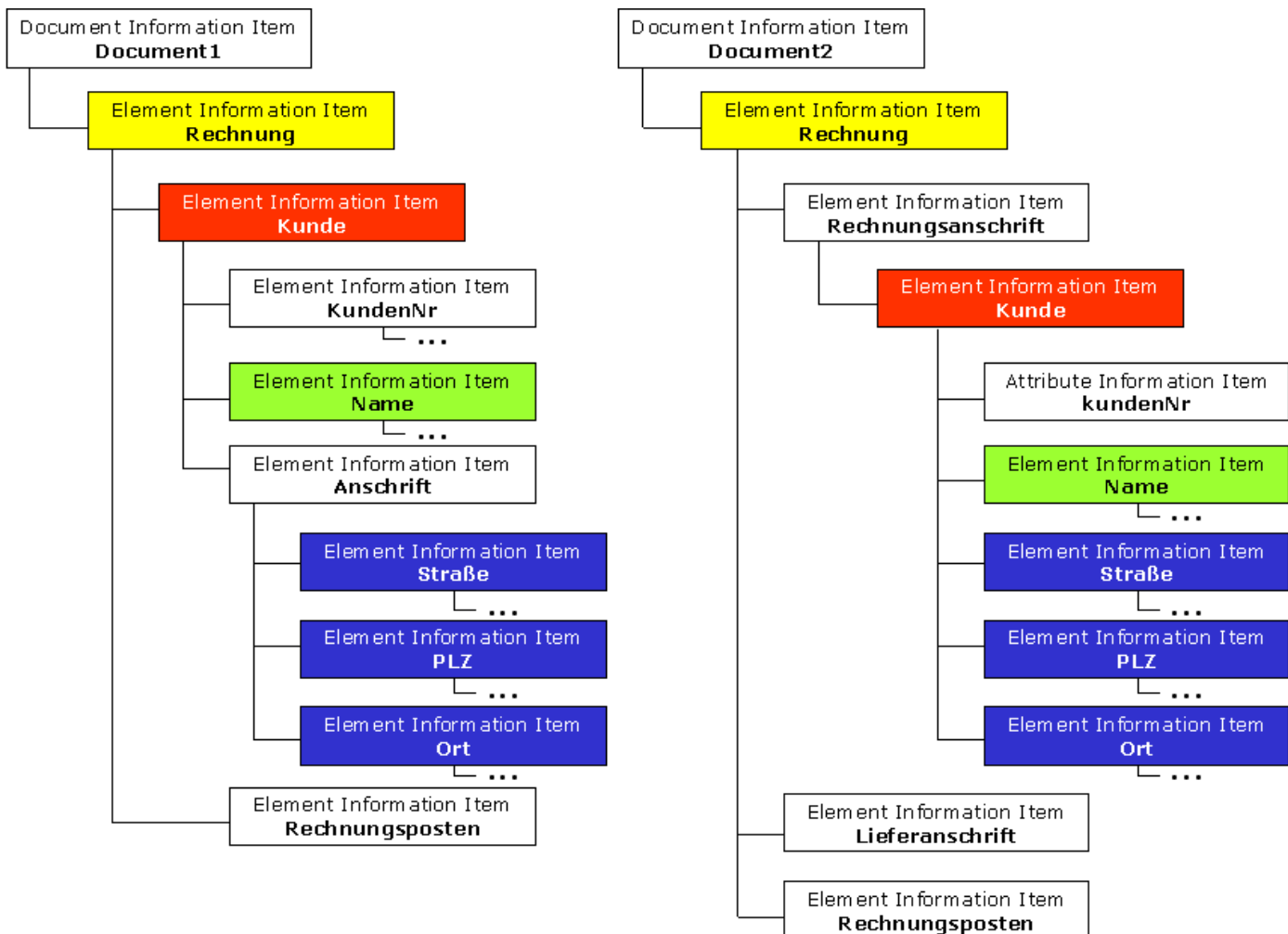


Abbildung 9: Information Sets der beiden Beispieldokumente (click on image to enlarge!)

Die beiden Bäume mit Information Set-Ausprägungen zeigen die Struktur der Beispieldokumente. Dabei sind Knoten die den selben Inhalt repräsentieren mit identischen Farben unterlegt, unabhängig davon um welchen Knotentyp es sich handelt. Die Character Information Item Knoten wurden aus Übersichtlichkeitsgründen weggelassen und durch Punkte angedeutet, sie sind jedoch für die vorliegende Betrachtung nicht von Interesse.

Einige der Elemente und Attribute werden in beiden Dokumenten mit gleichen Inhalten verwendet; z.B. Name, Ort oder PLZ. Dies äußert sich in identischen Teilbäumen unterhalb der Information Set-Knoten welche diese XML-Elemente repräsentieren. Hieraus läßt sich ableiten, daß die beiden vorgestellten Sprachen an den genannten Stellen keine strukturelle Differenz aufweisen. Dagegen unterscheiden sich die Kindknoten der Elemente Rechnung und Kunde hinsichtlich ihrer Struktureigenschaften. So folgt im ersten Beispieldokument auf das Rechnung-Element direkt der Kunde, während im zweiten XML-Dokument zunächst ein Element mit dem Namen

Rechnungsanschrift erwartet wird.

Dergleichen gilt für die Kindelemente des Kunden. Im zweiten Beispieldokument wird die diesem Element untergeordnete Kundennummer durch ein Attribut (kundenNr) dargestellt. Dagegen codiert das erste Beispiel diese Information direkt in den Elementinhalt.

Solange die beiden Dokumente in unterschiedlichen Anwendungswelten (Unternehmen o. ä.) verwendet werden, ist der gewählte Ansatz nicht problematisch. Bedenklich wird er jedoch in mindestens zweierlei Hinsicht:

Zunächst bei der „Mischung“ der beiden Dokumente. Dieser Wunsch tritt bei praktischen Problemstellungen häufig auf, wenn es um die Übernahme von XML-codierten Daten in ein anderes XML-Dokument geht. In der Konsequenz folgt das entstehende Zieldokument nicht mehr den Strukturierungsregeln eines der Ausgangsdokumente; mithin entsteht eine neue Dokumentstruktur, deren Regeln nicht explizit dokumentiert sind.

Eine weitaus größere Herausforderung stellt die Zusammenfassung und Veröffentlichung von XML-Strukturen in sog. Schemabibliotheken oder Datenbanken dar. Hier werden zwar die Dokumente nicht vereinigt, jedoch offenbart sich die gleiche Anwendungsdomäne (z.B. Rechnungsverwaltung, Stücklisten, Produktstrukturen) als problematisch, da sie die XML-Strukturen in direkte Konkurrenz treten läßt. In Zeiten immer stärker werdenden ökonomischen Flexibilisierungsdruckes erweist sich dies als äußerst kontraproduktiv, im Hinblick auf eine angestrebte Standardisierung. Die offene Konkurrenz verschiedener *Dialekte* innerhalb einer Domäne verzögert damit oft die Entscheidung zum Einsatz eines Sprachformates.

Einen anderen interessanten Anwendungsfall stellt der ausdrückliche Wunsch nach der Einbettung fremder Sprachelemente dar. Diese Form der Wiederverwendung knüpft an das durch öffentlich verfügbare XML-Formate eröffnete Anwendungsfeld an. Da nicht in jedem Fall ein alle Anforderungen erfüllendes existierendes XML-Format ermittelt werden kann, jedoch verschiedene vorhandene Formateile des gewünschten Umfangs abdecken, entsteht der Wunsch nach einer selektiven Weiterverwendung. Ein bekanntes Beispiel bilden Freitexte in beliebigen XML-Sprachen, welche auf Teile des (X)HTML-Sprachumfangs zurückgreifen. Gleichzeitig ist damit die Semantik der Elemente durch den zugehörigen W3C-Standard festgelegt. XHTML selbst stellt ein interessantes Anwendungsbeispiel für die gemeinsame Verwendung verschiedener XML-Sprachen in einem Dokument dar. So können Web-Seiten neben den bekannten Textstrukturen (XHTML) auch mathematische Symbole und Formeln (in der XML-Sprache [MathML](#)) und Vektorgraphiken (in der XML-Sprache [SVG](#)) enthalten.

Als Nebeneffekt der Wiederverwendung existierender XML-Sprachen verringern sich mögliche Fehlerquellen, was in der Konsequenz zur Erhöhung der Qualität der entstehenden Sprachen führt.

Zusammenfassend lassen sich die (Hinter-)Gründe der Namensraumeinführung wie folgt darstellen:

- Wiederverwendung bestehender (fremder) XML-Strukturen in eigenen Dokumenten.
- Wunsch nach breiteren Standards.
- Verringerung des Designaufwandes.
- Nutzung bereits gesammelter Designerfahrung.
- Zusammenführung verschiedener XML-codierter Inhalte (*heterogeneous content syndication*).

Definition 7: Namensräume



XML-Namensräume stellen eine XML-basierte Syntax zur Verfügung um Element- und Attributnamen eines Vokabulars eindeutig zu identifizieren und so Bedeutungsüberschneidungen durch gleichbenannte Elemente- oder Attribute in zu unterscheidenden Vokabularen auszuschließen. XML-Namensräume bilden damit die notwendige Voraussetzung zur freien dezentralen Entwicklung eigener Vokabulare ohne die Möglichkeit einer späteren Syndikatisierung zu verlieren.

Konzept der Namensräume:

Die Recommendation [Namespaces in XML](#) definiert die Syntax und Semantik der Namensräume. Ihr Konzept wurde rund ein Jahr nach Verabschiedung der ersten XML-Version eingeführt. Daher wurde der Kompatibilität mit bereits existierenden XML-Dokumenten große Priorität eingeräumt.

Grundidee der Namensräume ist es, die Element- und Attributnamen dergestalt zu erweitern, daß (auch nach Vereinigung beliebiger Dokumente wieder) eineindeutige Bezeichner entstehen. Dies könnte durch anwenderdefinierte Erweiterungen geschehen, sie trügen jedoch wiederum die Gefahr in sich, daß sie unbeabsichtigt mehrfach benutzt würden.

Daher scheidet der unkoordinierte Einsatz solcher Namenserverweiterungen aus. Jegliche Koordination bedingt jedoch inhärent eine zentrale Vergabestelle zur Registrierung der vergebenen Namen, die über die Eindeutigkeit wacht und Mehrfachnutzungen unterbindet.

Die Einführung einer solchen Stelle hätte jedoch einen unüberschaubaren Verwaltungsaufwand bedeutet, den das W3C nicht zu leisten im Stande wäre. Man nehme nur als Vergleich das Vergabeverfahren von Einträgen des Internet Domain Name Systems (DNS), welches bereits dezentral durch die einzelnen nationalen Domain-Registrars gehandhabt wird. Der dort anzutreffende Aufwand hätte sich für XML-Namensräume potenziert, legt man pro Domainadresse

mehrere Namensräume zugrunde.

Ziel des W3C war es, durch die Namensräume einen gleichermaßen mächtigen als auch leicht zu handhabenden und zu administrierenden Identifikationsmechanismus zu etablieren. Offenkundig wird diesem Anspruch nur ein (überwiegend) dezentraler, aber dennoch die Eineindeutigkeit garantierender, Ansatz gerecht.

Diesen Anforderungen genügt das aus [IETF RFC 2396](#) bekannte Namensschema der *Uniform Resource Identification* (URI) (später aktualisiert in [IETF RFC 2732](#)). Es kombiniert zentrale und dezentrale Elemente in der Handhabung, und ermöglicht so -- trotz Existenz und Pflege einer zentralen Registratur -- größtmögliche Flexibilität in der Anwendung. Der bekannteste Einsatz von URI-Namen ist der im World-Wide-Web allgegenwärtige *Uniform Ressource Locator* (URL) ([IETF RFC 1738](#)); einer Untermenge der URI.

Die zentrale Komponente findet sich im Domainnamen verwirklicht. Er ist entweder durch die IP-Adresse (konkret: IPv4-Adresse; im Falle des RFC 2732: der IPv6-Adresse) oder deren literaler Repräsentation gegeben. Unterhalb der Domänebene kann durch deren Verwalter eine beliebige Strukturierung vorgenommen werden. Die verschiedenen Ebenen werden dabei durch ISO-10646/ASCII #x2F „/“ voneinander abgetrennt.

Wie auch bereits bei URLs notwendig, ist das Schema (*URI scheme*) (z.B. http) zwingend mitanzugeben.

Trotz der Möglichkeit XML-Namensräume durch URLs zu identifizieren handelt es sich dabei nicht die Bezeichnung einer Internetquelle. Die verwendete Zeichenkette dient ausschließlich Benennung der im Namensraum versammelten XML [Element Information Items](#) und [Attribute Information Items](#).

Die Auflösung des Namensraumbezeichners durch einen XML-Prozessor ist nicht vorgesehen.

Nachfolgend ist die in definierte Syntax einer URI wiedergegeben. Sie wurde behutsam an die in der XML-Spezifikation verwendete BNF-Notation ([In XML-Spezifikation nachschlagen](#)) angepaßt, ohne jedoch die Produktionen in ihrer Struktur zu verändern.

```
[URI1]  URI-reference ::= (absoluteURI | relativeURI)? ("#" fragment)?
[URI2]  absoluteURI   ::= scheme ":" ( hier_part | opaque_part )
[URI3]  relativeURI   ::= ( net_path | abs_path | rel_path ) [ "?" query ]
[URI4]  hier_part     ::= ( net_path | abs_path ) ("?" query)?
[URI5]  opaque_part   ::= uric_no_slash uric?
[URI6]  uric_no_slash ::= unreserved | escaped | ";" | "?" | ":" | "@" |
                        "&" | "=" | "+" | "$" | ",",
[URI7]  net_path      ::= "//" authority abs_path?
[URI8]  abs_path      ::= "/" path_segments
[URI9]  rel_path      ::= rel_segment abs_path?
[URI10] rel_segment   ::= (unreserved | escaped |
                        ";" | "@" | "&" | "=" | "+" | "$" | ",") +
[URI11] scheme        ::= alpha (alpha | digit | "+" | "-" | "." ) *
[URI12] authority     ::= server | reg_name
[URI13] reg_name      ::= ( unreserved | escaped | "$" | "," |
                        ";" | ":" | "@" | "&" | "=" | "+" ) +
[URI14] server        ::= ((userinfo "@")? hostport)?
[URI15] userinfo      ::= ( unreserved | escaped |
                        ";" | ":" | "&" | "=" | "+" | "$" | ",") *
[URI16] hostport      ::= host (":" port)?
[URI17] host          ::= hostname | IPv4address
[URI18] hostname      ::= ( domainlabel "." ) * toplabel (".")?
[URI19] domainlabel   ::= alphanum | alphanum * ( alphanum | "-" ) alphanum
[URI20] toplabel      ::= alpha | alpha (alphanum | "-" ) * alphanum
[URI21] IPv4address   ::= digit+ "." digit+ "." digit+ "." digit+
[URI22] port          ::= digit*
[URI23] path          ::= (abs_path | opaque_part)?
[URI24] path_segments ::= segment ("/" segment)*
[URI25] segment       ::= pchar* (";" param)*
[URI26] param         ::= pchar*
[URI27] pchar         ::= unreserved | escaped |
                        ":" | "@" | "&" | "=" | "+" | "$" | ",",
```

```

[URI28] query      ::= uric*
[URI29] fragment   ::= uric*
[URI30] uric       ::= reserved | unreserved | escaped
[URI31] reserved   ::= ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+" |
                    "$" | ","
[URI32] unreserved ::= alphanum | mark
[URI33] escaped     ::= "%" hex hex
[URI34] hex        ::= digit | "A" | "B" | "C" | "D" | "E" | "F" |
                    "a" | "b" | "c" | "d" | "e" | "f"
[URI35] digit      ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
                    "8" | "9"
[URI36] uric_no_slash ::= unreserved | escaped | ";" | "?" | ":" | "@" |
                    "&" | "=" | "+" | "$" | ","

```

Die Produktionen alphanum, lowalpha sowie upalpha zur Konstruktion der alphanumerischen Namen wurden aus Übersichtlichkeitsgründen weggelassen.

Neben einigen anderen gängigen URI-Varianten stellt das nachfolgende Beispiel einige der möglichen syntaktisch korrekten URIs zusammen, die für die späteren Betrachtungen von Interesse sind.

Beispiel 9: Gültige URIs

- (1) <http://www.wi.fh-furtwangen.de>
- (2) <http://meinrechner.wi.fh-augsburg.de>
- (3) <mailto:mario@jeckle.de>
- (4) <ftp://ftp.shareware.com>
- (5) <http://www.jeckle.de/xml/vorlesung/script.htm#Namespaces>
- (6) [#EinfuehrungUndUeberblick](#)
- (7) <urn:oasis:names:specification:docbook:dtd:xml:4.1.2>
- (8) <urn:oid:1.3.6.1.2.1.27>
- (9) org.omg/standards/UML



Exkurs: URIs, URLs, URNs ...

Vielfach wird in der Praxis die Abgrenzung der im Internet gebräuchlichen Adressierungs- und Identifikationsmechanismen nicht trennscharf vollzogen. Darüberhinaus trat im Laufe der Entwicklung eine merkliche Bedeutungsverschiebung insbesondere zwischen der *Uniform Resource Identifikation* und den als WWW-Adressen genutzten *Uniform Resource Locators* ein.

Gegenwärtig wird die Begriffsabgrenzung wie in Abbildung 10 schematisch dargestellt vollzogen:

- *Uniform Resource Identification* (URI) dient als abstrakter Oberbegriff eineindeutig identifizierbarer Web-Ressourcen.

Konzeptionell sind URIs:

- über Zeit und Raum eindeutig
- für Menschen leicht zu merkend
- mit keinerlei Registrierungskosten verbunden
- unabhängig von der tatsächlichen Lokalisation der so identifizierten Ressource

Der URI-Raum zerfällt in die disjunkten Bezeichnerschemata URL, URN und URC.

- *Uniform Resource Location* (URL) bezeichnet den physischen Aufenthaltsort einer Ressource, etwa den Ablageort einer HTML-Seite.

Beispiele:

- <http://www.jeckle.de/vorlesung/xml/script.html>
- <http://www.wi.fh-furtwangen.de/>
- <mailto:mario@jeckle.de>
- <ftp://example.org/aDirectory/aFile>
- <news:comp.infosystems.www>
- <tel:+1-816-555-1212>
- <ldap://ldap.example.org/c=GB?objectClass=one>
- <urn:oasis:SAML:1.0>

- *Uniform Resource Name* (URN) bezeichnen den eineindeutigen Namen einer beliebigen Resource. Für die URN existiert kein Auflösungsmechanismus durch den die physische Lokation ermittelt

werden könnte. URNs dienen daher ausschließlich der eindeutigen Benennung!

Syntaktisch folgt auf das definiert Kürzel urn eine Zeichenkette welche eine Weiterklassifikation der Ressource gestattet. Hierdurch wird eine weitere Partitionierung des URN-Raumes erzielt. Diese *namespace ID* genannten Zeichenketten unterliegen einem globalen Registrierungszwang um ihre Eindeutigkeit zu gewährleisten. Diese Unterstrukturierungen sind in der Abbildung als ns₁ bis ns₃ benannt.

Beispiele:

- urn:oasis:names:specification:docbook:dtd:xml:4.1.2
- urn:oid:1.3.6.1.2.1.27
- **Uniform Resource Citation (URC)** schlägt die Brücke zwischen Lokationsbezeichnung und reiner Benennungskonvention. Eine URC verweist in eine Metadatenstruktur welche die physischen Ressourcen-Aufenthaltsorte katalogisiert. Dieser URI-Typ ist jedoch gegenwärtig kaum verbreitet.

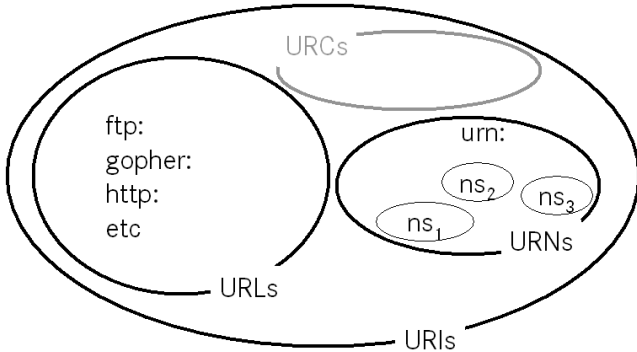


Abbildung 10: Die Definitionen der verschiedenen URI-Typen im Zusammenhang (click on image to enlarge!)



Web-Referenzen 4: Weiterführende Links

- [URIs, URLs, and URNs: Clarifications and Recommendations](#)
- [The Anatomy of an URL](#)

Verwendung von Namensräumen:

Am naheliegendsten wäre nach der Zielsetzung der Verwendung von URIs zur eindeutigen Benennung von XML-Element- und Attributnamen, die URI direkt vor dem XML-Bezeichner zu platzieren, evtl. separiert durch ein Trennsymbol wie den Doppelpunkt „:“. Hieraus entstünden dann, auf jeden Fall eindeutige, Element- und Attributnamen wie beispielsweise für das [erste Beispieldokument](#) dieses Kapitels (die URI `http://www.example.com/sales` werde zur Identifizierung verwendet):

```
(1)<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
(2)  <http://www.example.com/sales:Rechnung>
(3)      <http://www.example.com/sales:Kunde>
(4)          <http://www.example.com/sales:KundenNr>4711</http://www.
example.com/sales:KundenNr>
(5)              <http://www.example.com/sales:Name>Max Mustermann</http://www.
example.com/sales:Name>
(6)                  <http://www.example.com/sales:Anschrift>
(7)                      <http://www.example.com/sales:Straße>Musterplatz 1</
http://www.example.com/sales:Straße>
(8)                          <http://www.example.com/sales:PLZ>12345</http://www.
example.com/sales:PLZ>
(9)                              <http://www.example.com/sales:Ort>Musterstadt</http://
www.example.com/sales:Ort>
(10)                                  </http://www.example.com/sales:Anschrift>
(11)                                      </http://www.example.com/sales:Kunde>
(12)  <http://www.example.com/sales:Rechnungsposten>
(13)      ...
(14)  </http://www.example.com/sales:Rechnungsposten>
(15)</http://www.example.com/sales:Rechnung>
```

Bei entsprechender Nachbearbeitung des zweiten Beispieldokumentes mit einem anderen URI-identifizierten Namensraum, entstehen eindeutige Element- und Attributnamen, die nicht mehr kollidieren.

Jedoch verstößt diese Lösung gegen die in [Produktion 5](#) der XML-Spezifikation formulierte

syntaktische Einschränkung. Sie erlaubt das in URIs elementare Pfadtrennersymbol („/“) (aus den URI-Produktionen [8](#), [24](#) und [31](#)) nicht in XML-Namen (#x2F findet sich nicht in den in [Produktion 85](#) aufgeführten Unicode-Blöcken).

Die Integration der Namensräume auf diesem Weg hätte daher eine Modifikation der XML-Spezifikation nach sich gezogen. Diese erweiternde Aufweichung der zugelassenen Namen für Elemente und Attribute hätte jedoch mit der Kompatibilität zu SGML gebrochen, und somit eine der Grundforderungen der XML-Entwicklung verletzt.

Darüberhinaus ist die Spezifikation vollständiger URIs für Menschen „unhandlich“ und reduziert die Lesbarkeit der entstehenden XML-Dokumente.

Als Ausweg und pragmatischer Kompromiß zwischen uneindeutigen Namenspräfixen und Lesbarkeit wurde daher ein zweistufiges Verfahren eingeführt. Es erlaubt die Zuordnung von URIs zu Präfixen. Dieser Vorgang wird als „Bindung“ bezeichnet.

Diese Präfixe können Attributen oder Elementen vorangestellt werden, um sie in bestimmte Namensräume zu übernehmen.

Für die Präfixe gelten dieselben Bildungsgesetze wie für die Element- und Attributnamen. Im Einzelnen legt die *Namespace Recommendation* fest: (im XML-Namespaces-Dokument [nachschaugen](#))

```
[NS7] Präfix      ::= NCName
[NS4] NCName      ::= (Letter | '_' ) (NCNameChar)*
[NS5] NCNameChar ::= Letter | Digit | '.' | '-' | '_'
                  | CombiningChar
                  | Extender
```

Anmerkung: Die rechten Seiten der Produktionen beziehen sich entweder auf die dargestellten Definitionen des Namespace-Standards oder auf Syntaxregeln der XML-Recommendation.

Die Bindung einer URI an ein -- gemäß [Produktion NS7](#) frei wählbares -- Präfix geschieht durch das reservierte Attribut `xmlns`.

Die Syntax hierfür wird mit

```
[NS2] PrefixedAttName ::= 'xmlns:' NCName
```

angegeben.

Nach der Bindung der URI an das Präfix kann dieses jedem Element oder Attribut vorangestellt werden, um es in den Namensraum zu übernehmen.

Hierdurch verändert sich die [Produktion Name aus der XML-Spezifikation](#) zum *qualifizierten Namen*, der durch die Voranstellung des Präfixes entsteht. Der rechts vom trennenden Doppelpunkt folgende Elementname stellt den lokalen Namen (innerhalb des Namensraumes dar). Dieser lokale Name darf *keinen* Doppelpunkt mehr enthalten; insofern schränkt [Produktion NS8](#) in Verbindung mit [NS4](#) die Festlegung der [Produktion 5](#) der XML-Spezifikation ein.

```
[NS6] QName       ::= (Präfix ':' )? LocalPart
[NS8] LocalPart   ::= NCName
```

Während der Verarbeitung eines XML-Dokuments, das Namensräume nutzt, ersetzt ein XML-Prozessor jedes Auftreten eines deklarierten Präfixes transparent durch die gebundene URI. Prozessoren, welche die Namensraum-Spezifikation unterstützen, werden als *namespace aware* bezeichnet. Alle anderen Prozessoren treffen die durch NS6 eingeführte Unterscheidung zwischen *Präfix* und *LocalPart* eines qualifizierten Namens nicht und betrachten die Kombination aus Präfix und Element- bzw. Attributnamen als Bezeichner. Die Präfix-URI-Bindung durch das `xmlns:...-Attribut` wird hierbei als gewöhnliches XML-Attribut betrachtet und führt daher zu keinen Validierungsfehlern. (Die Einschränkung der [Produktion 5](#), ein Name dürfe nicht mit der Zeichenfolge (('X'|'x') ('M'|'m') ('L'|'l')) beginnen, stellt in der XML-Spezifikation lediglich einen Hinweis dar.)

Anmerkung: Auf Webseiten und in Mailinglisten finden sich manchmal Formulierungen der Struktur `{namespaceName}elementName` (z.B. `{http://www.w3.org/2001/XMLSchema}element` oder `{http://www.w3.org/1999/XSL/Transform}template`).

Hierbei handelt es sich um eine zwar geläufige, aber *nicht spezifikationskonforme Schreibweise*! Sie dient lediglich dazu, das prinzipiell beliebig wählbare Präfix einzusparen und den gewählten Namensraum hervorzuheben.

Strukturen dieses Stils sind jedoch keine gültigen XML-Dokumente!

Angewendet auf das betrachtete Beispiel läßt sich die URI `http://www.example.com/sales` an das

Präfix myNS1 binden. Diese Bindung steht im definierenden Element (local name: rechnung) und allen untergeordneten zur Verfügung.

Beispiel 10: Dokument mit W3C-konformen Namensräumen



```
(1)<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
(2)<myNS1:Rechnung xmlns:myNS1="http://www.xyz.com/sales">
(3)  <myNS1:Kunde>
(4)    <myNS1:KundenNr>4711</myNS1:KundenNr>
(5)    <myNS1:Name>Max Mustermann</myNS1:Name>
(6)    <myNS1:Anschrift>
(7)      <myNS1:Straße>Musterplatz 1</myNS1:Straße>
(8)      <myNS1:PLZ>12345</myNS1:PLZ>
(9)      <myNS1:Ort>Musterstadt</myNS1:Ort>
(10)    </myNS1:Anschrift>
(11)  </myNS1:Kunde>
(12)  <myNS1:Rechnungsposten>
(13)    <!-- ... -->
(14)  </myNS1:Rechnungsposten>
(15)</myNS1:Rechnung>
```

Download des Beispiels

Hinweis: Für das Attribut xmlns kann keine Namensraumdeklaration angegeben werden; es ist spezifikationsgemäß an keinen Namensraum gebunden.

Die Deklaration des Namensraumes mit der Präfixbindung kann auf beliebige hierarchisch höhergeordnete Elemente ausgelagert werden. In der Praxis hat es sich aus Übersichtlichkeitsgründen durchgesetzt, alle in einem XML-Dokument benutzten Namensräume mit ihren Präfixen zu Beginn des Dokuments im Wurzelement zu definieren.

Das nachfolgende Beispiel zeigt dies anhand eines XHTML-Dokuments, das neben Elementen der Hypertextsprache auch mathematische Formeln und Vektorgraphiken enthält.

Beispiel 11: Ein XHTML-Dokument mit MathML- und SVG-Inhalten



```
(1)<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
(2)<xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml" xmlns:mml="http://www.w3.org/TR/REC-MathML" xmlns:svg="http://www.w3.org/2000/svg">
(3)  <xhtml:head>
(4)    <xhtml:title>XHTML Dokument, mit MathML- und SVG-Inhalten</xhtml:
title>
(5)  </xhtml:head>
(6)  <xhtml:body>
(7)    <xhtml:h1>Eine Überschrift</xhtml:h1>
(8)    <mml:math>
(9)      <mml:mrow>
(10)        <mml:mi>x</mml:mi>
(11)        <mml:mo>=</mml:mo>
(12)        <mml:mfrac>
(13)          <mml:mrow>
(14)            <mml:mrow>
(15)              <mml:mo>-</mml:mo>
(16)              <mml:mi>b</mml:mi>
(17)            </mml:mrow>
(18)            <mml:mo>&PlusMinus;</mml:mo>
(19)            <mml:msqrt>
(20)              <mml:mrow>
(21)                <mml:msup>
(22)                  <mml:mi>b</mml:mi>
(23)                  <mml:mn>2</mml:mn>
(24)                </mml:msup>
(25)                <mml:mo>-</mml:mo>
(26)                <mml:mrow>
(27)                  <mml:mn>4</mml:mn>
(28)                  <mml:
mo>&InvisibleTimes;</mml:mo>
(29)                  <mml:mi>a</mml:mi>
(30)                  <mml:
mo>&InvisibleTimes;</mml:mo>
(31)                  <mml:mi>c</mml:mi>
(32)                </mml:mrow>
(33)              </mml:mrow>
(34)            </mml:msqrt>
```

```

(35)                                </mml:mrow>
(36)                                <mml:mrow>
(37)                                <mml:mn>2</mml:mn>
(38)                                <mml:mo>&InvisibleTimes;</mml:mo>
(39)                                <mml:mi>a</mml:mi>
(40)                                </mml:mrow>
(41)                                </mml:mfrac>
(42)                                </mml:mrow>
(43)                                </mml:math>
(44)                                <svg:svg width="4cm" height="8cm">
(45)                                <svg:ellipse cx="2cm" cy="4cm" rx="2cm" ry="1cm"/>
(46)                                </svg:svg>
(47)                                </xhtml:body>
(48) </xhtml:html>

```

[Download des Beispiels](#)



Definition 8: Namensraumidentifikation

Jeder XML-Namensraum wird durch eine gültige URI identifiziert. Diese URI dient ausschließlich der Benennung, daher muß sie nicht auf eine gültige Ressource verweisen.

Überschreiben des Vorgabe-Namensraums:

Aus den Beispielen ist leicht ersichtlich, daß die explizite Angabe des definierten Präfixes für jedes Element eines Namensraumes platzraubend und für die Zuordnung aller Elemente eines Teilbaumes zum selben Namensraum redundant und -- wegen des zusätzlichen Spezifikationsaufwandes -- unpraktikabel ist. Die mehrmalige explizite redundante (identische) Angabe des identifizierenden Präfixes bildet zusätzlich noch eine potentielle Fehlerquelle hinsichtlich Übertragungsfehlern und reiner Tippfehler bei manuell erstellten XML-Dokumenten.

Eine einfache Kompaktifizierungsvariante greift auf die aus den Programmiersprachen geläufigen Regeln für Namensräume zurück. Dort beinhaltet ein explizit geöffneter Block alle enthaltenen Elemente bis zum Blockendesymbol und faßt sie so zu einem Gültigkeitsbereich zusammen. Dieses Prinzip läßt sich leicht auch auf XML-Dokumente, die immer eine streng hierarchische Baumstruktur aufweisen, anwenden.

Hierzu wird das `xmlns`-Attribut leicht modifiziert eingesetzt. Wird es *ohne nachfolgendes Präfix* und unter Weglassung des separierenden Doppelpunktes verwendet, so definiert es einen *Vorgabename* (*default namespace*). Dieser umfaßt neben dem Element, welches das Attribut beinhaltet, auch alle Kindelemente. Eine Ausnahme hiervon bilden untergeordnete Elemente, die explizit durch Präfix oder Redefinition des Vorgabename-Namensraumes einem anderen Namespace zugeordnet werden.

Das nachfolgende Beispiel zeigt dies für das [bereits mit Namenräumen versehene Rechnungsdokument](#)

Beispiel 12: Rechnungsdokument mit überschriebenem Vorgabename

```

(1) <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
(2) <Rechnung xmlns="http://www.xyz.com/sales">
(3)   <Kunde>
(4)     <KundenNr>4711</KundenNr>
(5)     <name>Max Mustermann</Name>
(6)     <Anschrift>
(7)       <Straße>Musterplatz 1</Straße>
(8)       <PLZ>12345</PLZ>
(9)       <Ort>Musterstadt</Ort>
(10)    </Anschrift>
(11)  </Kunde>
(12)  <Rechnungsposten>
(13)    <!-- ... -->
(14)  </Rechnungsposten>
(15) </Rechnung>

```



[Download des Beispiels](#)

Durch die Definition des Vorgabename-Namensraumes für das Element `rechnung` und all dessen Kindelemente wird derselbe Effekt erreicht wie durch die Präfixangabe im [vorangegangenen Beispiel](#).

Diese Schreibweise stellt lediglich eine Abkürzung der expliziten Qualifizierung jedes einzelnen XML-Namens dar. Insbesondere führt die mehrmalige Redefinition des Vorgabennamensraumes *nicht* zu kaskadierten Namensräumen. Jeder Namensraum ist von allen umgebenden unabhängig definiert. So kann das Dokument des [XHTML-Beispiels](#) auch dahingehend verändert werden, daß die Namensräume erst an der Stelle im Dokument deklariert werden, an der sie auch benötigt werden.

Beispiel 13: Ein XHTML-Dokument mit MathML- und SVG-Inhalten, unter Verwendung überschriebener Vorgabennamensräume

```
(1)<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
(2)<html xmlns="http://www.w3.org/1999/xhtml">
(3)  <head>
(4)    <title>XHTML Dokument, mit MathML- und SVG-Inhalten</title>
(5)  </head>
(6)  <body>
(7)    <h1>Eine Überschrift</h1>
(8)    <math xmlns="http://www.w3.org/1998/Math/MathML">
(9)      <mrow>
(10)        <mi>x</mi>
(11)        <mo>=</mo>
(12)        <mfrac>
(13)          <mrow>
(14)            <mrow>
(15)              <mo>-</mo>
(16)              <mi>b</mi>
(17)            </mrow>
(18)            <mo>+</mo>
(19)            <msqrt>
(20)              <mrow>
(21)                <msup>
(22)                  <mi>b</mi>
(23)                  <mn>2</mn>
(24)                </msup>
(25)                <mo>-</mo>
(26)                <mrow>
(27)                  <mn>4</mn>
(28)                  <mo>&#160;</mo>
(29)                  <mi>a</mi>
(30)                  <mo>&#160;</mo>
(31)                  <mi>c</mi>
(32)                </mrow>
(33)              </mrow>
(34)            </msqrt>
(35)          </mrow>
(36)          <mrow>
(37)            <mn>2</mn>
(38)            <mo>&#160;</mo>
(39)            <mi>a</mi>
(40)          </mrow>
(41)        </mfrac>
(42)      </mrow>
(43)    </math>
(44)    <svg xmlns="http://www.w3.org/2000/svg" xmlns:svg="http://www.w3.
org/2000/svg" svg:width="4cm" svg:height="8cm">
(45)      <ellipse cx="2cm" cy="4cm" rx="2cm" ry="1cm"/>
(46)    </svg>
(47)  </body>
(48)</html>
```

[Download des Beispiels](#)

Die Namensraumpräfixe können durch den Anwender frei vergeben werden. Sie dienen lediglich der abkürzenden Schreibweise und sind für die Namensraumauflösung unerheblich.

Daher werden zwei Elemente oder Attribute als gleich betrachtet, wenn sie lexikalisch in Namen und Namensraumidentifizier übereinstimmen. Hierbei ist es unerheblich, ob der Namensraum explizit durch Präfixangabe oder durch Überschreiben des Vorgabennamensraumes definiert wurde.

Die Elemente der XML-Dokumente aus den Beispielen 14 und 15 befinden sich alle ausnahmslos im Namensraum <http://www.example.com>.

Beispiel 14: Namensraumpräfixe 1



```
(1)<abc:ElementA xmlns:abc="http://www.example.com"
(2)                                xmlns:xyz="http://www.example.com">
(3)    <ElementB xmlns="http://www.example.com">
(4)        <ElementC/>
(5)    </ElementB>
(6)    <xyz:ElementB>
(7)        <abc:ElementC/>
(8)    </xyz:ElementB>
(9)</abc:ElementA>
```

[Download des Beispiels](#)

Beispiel 15: Namensraumpräfixe 2



```
(1)<ElementA xmlns="http://www.example.com"
(2)                                xmlns:myNamespace="http://www.example.com">
(3)    <foo:ElementB xmlns:foo="http://www.example.com">
(4)        <myNamespace:ElementC/>
(5)    </foo:ElementB>
(6)    <ElementB xmlns="http://www.example.com">
(7)        <myNamespace:ElementC/>
(8)    </ElementB>
(9)</ElementA>
```

[Download des Beispiels](#)

Die Abbildung zeigt das Beispieldokument in der Darstellung des W3C-Browsers [Amaya](#).

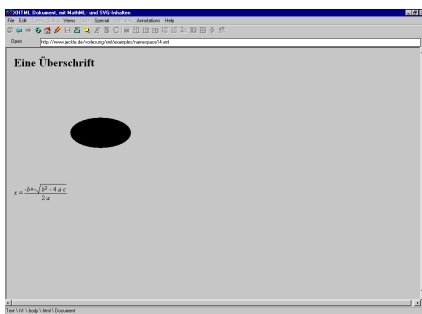


Abbildung 11: Screenshot im Browser (click on image to enlarge!)

Im Beispieldokument wird der Vorgabennamespace dreimal, entsprechend der verschiedenen verwendeten XML-Sprachen, neu gesetzt. So wird auf html und alle direkt untergeordneten Elemente der URI-identifizierte Namespace <http://www.w3.org/1999/xhtml> angewendet. head, title und body sowie dessen Kindelemente finden sich demnach, da sie keinen eigenen Namespace definieren, ebenfalls im so definierten Vorgabennamespace.

mrow als hierarchisch tieferstehendes Element redefiniert den Namespace zu <http://www.w3.org/TR/REC-MathML>. Daher werden das Element mrow sowie all dessen Kindelemente (im Beispiel: ellipse) auch diesem zugeordnet.

Die Attribute width, height, cx, ... verfügen über kein explizites Namespacepräfix und sind daher dem leeren Namespace zugeordnet.

Auf den MathML-Namespace folgend wird der Vorgabennamespace zu <http://www.w3.org/2000/svg> redefiniert. Auch hier gelten dieselben Regeln, d.h. der überschriebene Vorgabennamespace erstreckt sich auf alle Kindelemente.

Mit dem schließenden Tag svg endet auch dessen Namespace. Alle folgenden Elemente befinden sich wieder im umgebenden Namespace, der zu Beginn des Dokuments mit <http://www.w3.org/1999/xhtml> festgelegt wurde.

Die nachfolgende Graphik stellt die Namensräume nochmals farblich hervorgehoben dar. Ein [weiteres Beispiel](#) findet sich in der Namespace-Recommendation.

Namensräume für Attribute:

Abweichend von der Mimik für Elemente, dort wirkt sich ein überschriebener Vorgabennamespace auch immer auf die Kindelemente aus, wird eine Namensraumdeklaration auf Elementebene nicht auf Attribute propagiert.

Diese Festlegung der Spezifikation mag insbesondere unter Kenntnis der Baumstruktur der Infosets, welche Attribute und Elemente gleichermaßen als Kindknoten der beherbergenden Elementinformationseinheit darstellt, verwundern. Eine mögliche Begründung dieser Asymmetrie mag in der besonderen Rolle der Attribute zur Informationsdarstellung liegen. So wird teilweise damit argumentiert, daß Attribute üblicherweise unabhängig vom aktuell umgebenden Element sein sollten und daher nur zur Darstellung von Daten herangezogen werden sollten, die nicht über einen direkten Bezug zum sie umgebenden Element verfügen.

In der Konsequenz müssen Attribute immer explizit mit einem Namensraumpräfix versehen werden, um sie einem Namensraum zuzuordnen.

Beispiel 17 zeigt die Anwendung der Namensräume auf Attribute. So befinden sich weder das Attribute att1 des Elements ElementB, noch dasjenige von ElementD in einem Namensraum. Das mit dem Wert XYZ versehene Attribut att2 des Elements ElementC wird hingegen -- aufgrund des explizit angegebenen Präfixes -- dem Namensraum `http://www.example.com/NS2` zugeordnet. Ferner illustriert ElementC die Rolle der Namensräume als Bestandteil des identifizierenden Namens von Elementen und Attributen. Aufgrund der Interpretation des Namensraumes als Benennungsbestandteil darf das att2 benannte Attribut mehrfach auftreten, da die Zuhilfenahme des Namensraumes die eindeutige Identifikation gestattet.

Beispiel 17: Namensräume für Attribute

```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<Wurzelelement>
(3)    <ElementA xmlns:NS1="http://www.example.com/NS1" xmlns:NS2="http://www.
example.com/NS2">
(4)        <ns2:ElementB att1="...">
(5)            <ElementD att1="..." xmlns="http://www.example.com/NS3">
(6)                <ElementC att2="ABC" NS2:att2="XYZ"/>
(7)            </ElementD>
(8)        </ns2:ElementB>
(9)    </ElementA>
(10)</Wurzelelement>
```

**Definition 9: Namensraumvererbung**

Namensräume, die durch Überschreiben des Vorgabennamesraumes zugewiesen werden wirken sich ausschließlich auf Elemente und deren direkte oder transitive Kindelemente aus, sofern diese den Namensraum nicht wieder verändern.

Namensräume, die durch explizite Präfixangabe zugewiesen werden, wirken sich ausschließlich auf dasjenige Element aus vor dessen Name das Präfix plazierte ist. Namensräume für Attribute werden ausnahmslos durch explizite Präfixangabe festgelegt und gelten ausschließlich für das Attribut selbst.

**Kompatibilität zu älteren Dokumenten:**

Elemente, für die weder ein expliziter Namensraum durch Präfix definiert ist, noch ein Namensraum von einem Elternelement übernommen werden kann, sind einem leeren Namensraum zugeordnet; konzeptionell entspricht dies einem NULL-Präfix.

Somit befinden sich alle Elemente, die keinem Namensraum angehören, automatisch in einem gemeinsamen Namensraum, der an keine URI gebunden ist.

Zusammenfassend gelten somit folgende Prinzipien:

- Jede beliebige URI kann an eigendefinierte Präfixe gebunden werden. Insbesondere kann dieselbe URI an verschiedene Präfixe gebunden werden, und dasselbe Präfix in verschiedenen Kontexten für verschiedene URIs stehen.
- Jedes Element übernimmt den überschriebenen Vorgabennamespace seines Elternelements, sofern es keinen eigenen definiert.
- Elemente oder Attribute ohne Namensraumzuordnung (die auch keine von ihrem hierarchisch höherstehenden Element übernehmen) befinden sich im Standardnamespace.
- Attribute ohne Namensraumpräfix befinden sich im leeren Namensraum.

Ausblick auf XML-Namensräume v1.1

Derzeit befindet sich ein [Arbeitsstand zur Weiterentwicklung der XML-Namensräume](#) innerhalb des W3Cs in der Diskussion. Dieses Dokument ist inhaltlich eng mit den Änderungen in der ebenfalls noch nicht fertiggestellten [Version 1.1 des XML-Kernstandards](#) verknüpft. So wird insbesondere auf die veränderte Behandlung von Zeichendaten aus dem Unicode-Standard der Version 3.1 Rücksicht

genommen. Zusätzlich wird die diese Fassung der XML-Namensraumspezifikation einen Mechanismus zur expliziten Aufhebung einer erfolgten Namensraumdeklaration vorsehen.

Die Berücksichtigung von Zeichen, die in XML v1.1 zugelassenen, deren Nutzung in den klassischen URIs nach [RFC 2396](#) bzw. [RFC 2732](#) jedoch untersagt ist, führt zur Einführung des neuen Begriffes des *Internationalized Resource Identifiers* (IRI). Diese Neuschöpfung stellt im Kern eine URI-Fassung dar innerhalb der Leerzeichen sowie diverse Sonderzeichen zulassen sind. Diese internationalisierten Identifikatoren werden durch einen im Spezifikationsentwurf festgelegten Algorithmus in syntaktisch korrekte URIs umgewandelt.

Beispiel 18 zeigt gültige IRIs und jeweils dahinter in Klammern angegeben die daraus resultierende URI-Darstellung.



Beispiel 18:

(1) `http://www.{iri-}example.com` (`http://www.%7Biri-%7Dexample.com`)
 (2) `mailto:marc léon@example.org` (`mailto:marc%20l%E9on@example.org`)

Zusätzlich führt Version 1.1 der Namensraumspezifikation die Möglichkeit ein Namensräume durch Bindung der leeren Zeichenkette explizit zurückzusetzen.

Beispiel 19: Aufheben von Namensraumdeklarationen

(1) `<?xml version="1.0" encoding="UTF-8"?>`
 (2) `<Adressen>`
 (3) `<table xmlns="http://www.w3.org/TR/REC-html40">`
 (4) `<tr>`
 (5) `<td>Name</td>`
 (6) `<td>Adresse</td>`
 (7) `</tr>`
 (8) `<tr>`
 (9) `<td>`
 (10) `<Vorname xmlns="">Max</Vorname>`
 (11) `<Nachname xmlns="">Mustermann</Nachname>`
 (12) `</td>`
 (13) `<td>`
 (14) `<Straße xmlns="">Musterstr. 1</Straße>`
 (15) `<PLZ xmlns="">12345</PLZ>`
 (16) `<Ort xmlns="">Musterstadt</Ort>`
 (17) `</td>`
 (18) `</tr>`
 (19) `</table>`
 (20) `</Adressen>`

Das Beispiel 19 zeigt die notwendigen Deklarationen zur Aufhebung der Vorgabenamensraumdefinition.

So wird zwar für das Element `table` und alle seine Kindelemente der Vorgabenamensraum auf `http://www.w3.org/TR/REC-html40` gesetzt, dies jedoch für die Kindelemente `Vorname`, `Nachname`, `Straße`, `PLZ` und `Ort` durch die Festlegung `xmlns=""` explizit für das jeweilige Element aufgehoben.

Web-Referenzen 5: Weiterführende Links

- [XML-Namespace Recommendation](#)
- [Namespace Recommendation in deutscher Übersetzung](#)
- [Namespace Tutorial @ Zvon.org](#)
- Tim Bray: [Namespaces by Example](#)
- Hintergrundartikel: [Namespaces in XML Adopted by W3C](#)
- (Tutorial) Simon St. Laurent: [Namespaces in XML](#)
- Roland Bourret: [XML Namespaces FAQ](#)



XML Schema

Neben den in der Vergangenheit zur Sprachdefinition verwendeten Document Type Definitions ist in jüngerer Zeit ein alternativer Ansatz in den Blickpunkt des Interesses gerückt: die XML-Schemasprachen.

Sie setzen die Emanzipation der Metasprache XML von ihrer Vorgängersprache SGML fort. Bereits in engem zeitlichem Bezug zur Veröffentlichung der XML-Recommendation wurde mit [XML Data](#) ein

erster Ansatz vorgestellt. In der Zwischenzeit fanden verschiedene konkurrierende Vorschläge ein breites Interesse. Übereinstimmende Zielsetzung aller verschiedenen vorgeschlagenen Schemasprachen ist die Schaffung eines Sprachdefinitionsmechanismus, der die Dokumenten-orientierten Strukturen und Inhaltsmodelle der DTD überwindet.

An die Spitze der Bemühungen setzte sich eine [Arbeitsgruppe des W3C](#) zur Definition einer XML-Schemasprache, unter Berücksichtigung der bekanntesten und verbreitetsten Vorschläge. Durch sie wurde im Mai 2001 der *XML Schema*-Standard des W3C veröffentlicht.

Der Begriff *Schema* ist der im Datenbankumfeld gebräuchlichen Terminologie entlehnt. Dort bezeichnet er Informations- oder Datenmodelle als Konstruktionsvorlage oder Dokumentation eines Datenbankdesigns. Hierzu muß ein Schema nicht unbedingt in einer graphischen Datenmodellierungssprache vorliegen, sondern kann beispielsweise auch die Tabellenstruktur einer relationalen Datenbank bezeichnen.

Zur Notwendigkeit einer Schemasprache:

Zum Zeitpunkt der Konzeption der Metasprache SGML war das Anwendungsfeld klar umrissen und im wesentlichen auf die Digitalisierung vormals papiergestützter Dokumentation festgelegt. Daraus erklärt sich auch die Mächtigkeit der Document Type Definition, der angebotenen Grammatiksprache zur Darstellung der Dokumentstrukturen.

Insbesondere war weder die Daten-orientierte Verwendung von SGML, noch die rund 30 Jahre später einsetzende Weiterentwicklung (eigentlich: Reduktion) zur eXtensible Markup Language abzusehen.

Die inzwischen eingesetzte breite Anwendung von [XML-Sprachen](#) zur Darstellung beliebiger Inhalte läßt jedoch die Beschränkungen und Unzulänglichkeiten des DTD-Mechanismus für diesen Anwendungen offenkundig werden.

Nachfolgend sind einige der durch Nutzung des DTD-Mechanismus zur Beschreibung Daten-intensiver Strukturen induzierten Einschränkungen zusammengestellt:

- Unzureichende Datentypunterstützung.
DTDs erlauben für Elemente nur die vier Inhaltsmodelle: *child elements*, *PCDATA*, *mixed content* sowie das leere Inhaltsmodell *EMPTY*.
- Unzureichende Strukturierungsunterstützung.
Zwar erlauben die Operatoren zur Steuerung der Auftrittshäufigkeit („?", „+“ und „*“) einzelner Kindelemente die Codierung beliebiger Kardinalitäten. Allerdings sind die hierfür anzuwendenden Prinzipien teilweise umständlich in der Schreibung, und daher fehlerträchtig. Die Lesbarkeit der entstehenden DTD wird entscheidend gesenkt.
- Keine Unterstützung von Wiederverwendbarkeit.
Während Elementstrukturen (zumindest) innerhalb der definierenden DTD beliebig wiederverwendet werden können, sind Attribute immer an das umgebende Element gebunden. Eine Nutzung in anderen Dokument-Typ-Definitionen als der definierenden ist nicht vorgesehen.
- Starres Typsystem.
Das angebotenen Typsystem kann durch den Anwender nicht erweitert werden.
- Keine Unterstützung von Namensräumen.
Namensräume können in der DTD nicht angegeben werden.
- Nur rudimentärer Referenzierungsmechanismus.
Die offerierten Verknüpfungsmechanismen sind ausschließlich Dokument-lokal möglich und gestatten keine Differenzierung hinsichtlich der Semantik des eindeutig identifizierten oder referenzierten Elements.
- DTD-Syntax ist nicht XML.
Die von SGML übernommene Syntax der DTD bildet eine eigene Sprache, die von Werkzeugen gesondert zu implementieren ist.

Technische Ansätze:

Prinzipiell lassen sich die in der Vergangenheit vorgeschlagenen Ansätze zur Definition einer Schemasprache in vier Kategorien unterscheiden:

1. Orientierung am bestehenden DTD-Mechanismus.
Erweiterungen des bestehenden Mechanismus um zusätzliche Sprachelemente.
2. Orientierung an der programmiersprachlichen Interpretation.
Versuch XML und ein Ausführungsmodell möglichst eng zu koppeln.
3. Orientierung an Wissensdarstellungen
Interpretation des Schemas einer XML-Sprache als Wissen über die Sprache.
4. XML-Sprachen zur Inhaltsbeschreibung.
Da XML i.A. zur Beschreibung beliebigster Informationen herangezogen werden kann, ist die Verwendung auch für die Beschreibung von XML-Strukturen denkbar.

Die naheliegendste Option dürfte die Erweiterung des bestehenden DTD-Sprachumfanges bilden. Durch geeignete Modifikationen und Ergänzungen ließen sich alle, mit Ausnahme der letzten, identifizierten Unzulänglichkeiten beheben.

Konzeptionell lassen sich zwei Erweiterungsvarianten aufzeigen. Zunächst die Möglichkeit, die XML-DTDs um Elemente der ursprünglichen SGML-DTD zu erweitern. In der Konsequenz nähert sich XML, positiv formuliert, wieder der Ausdrucksmächtigkeit der Ursprache SGML an. Negativ formuliert, kann jedoch XML auf diesem Wege niemals Inhaltsstrukturen ausdrücken, die nicht durch SGML ausdrückbar sind, da die Mächtigkeit des SGML-DTD-Mechanismus eine natürliche Obergrenze der Erweiterbarkeit darstellt. Zusätzlich ist anzumerken, daß ein solcher Ansatz der ursprünglichen Intention der XML-Entwicklung -- ein leichter einsetzbares SGML zu schaffen -- entgegenläuft.

Eine der bekannten Ideen zur Erweiterung des DTD-Mechanismus stellt [Datatypes for DTDs \(DT4DTD\)](#) dar.

Alternativ zur Erweiterung hin zur SGML-Mächtigkeit ließe sich der bestehende XML-DTD-Mechanismus um neue zusätzliche Konstrukte anreichern, die nicht Bestandteil der SGML-DTD-Syntax sind. Dieser Ansatz böte den Vorteil, den Vorgängerstandard nicht berücksichtigen zu müssen und beliebige Erweiterungen in Syntax und Semantik einbringen zu können. Allerdings würde damit eine zentrale Forderung der XML-Entwicklung, die sich bereits im [Abstract der XML-Recommendation](#) findet, nicht berücksichtigt: die Untermengenbeziehung zu SGML. Durch eine Erweiterung, welche über die SGML-Mächtigkeit hinausreicht, würden legale (*well formed* und sogar *valid*) XML-Dokumente entstehen, die keine gültigen SGML-Dokumentinstanzen wären.

Die nachfolgende Graphik veranschaulicht die beiden Erweiterungsoptionen und die Argumente der geführten Diskussion.

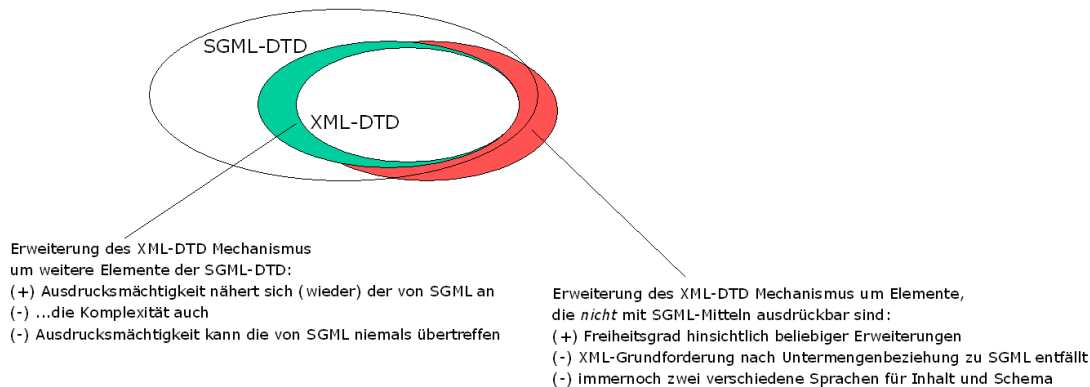


Abbildung 13: Optionen zur Erweiterung des bestehenden DTD-Mechanismus (*click on image to enlarge!*)

Die im zweiten Punkt angedeutete Umsetzung ist durch eine programmiersprachliche Verarbeitung der XML-Dokumente motiviert. Aus Sicht dieser Anwendungsfacette ist ein Schemamechanismus idealerweise so ausgelegt, daß er die transparente Umsetzung in Applikationsdatenstrukturen ermöglicht. Dahinter steht der Wunsch, den *impedance mismatch*, mithin den zu leistenden Abbildungsaufwand zwischen XML-Konstrukten und Datenstrukturen, möglichst gering zu halten. Beispielsweise greift der -- durch den Einsatz im e-Commerce-System der Firma [CommerceOne](#) bekannt gewordene -- Vorschlag [Schema for Object-Oriented XML \(SOX\)](#) zur Definition der notwendigen Semantik der angebotenen Schemaprimtiven auf die bekannte plattformunabhängige Programmiersprache [Java](#) zurück.

Die aktuelle Version der Schemasprache SOX, die zur Definition der XML-Sprache [xCBL](#) eingesetzt wird, findet sich unter [xCBL.org](#).

Der dritte technische Ansatz weist auf eine alternative Interpretation der XML-Grammatikstruktur hin. So spiegelt ein Schema auch immer *Wissen* über Struktur und Inhalt eines betrachteten Problembereichs wieder.

Der bekannteste Vorschlag -- die [Document Content Description \(DCD\)](#) -- nutzt zur Definition der Wissensstrukturen eines XML-Dokuments das [Resource Description Framework](#) (RDF) des World Wide Web Consortiums.

Der Ansatz hat sich durch Referenzimplementierungen durchaus als tragfähig und, wegen der RDF-basiertheit, als allgemein verwendbar erwiesen. Jedoch liegt hierin auch die offensichtlichste Limitierung. RDF als Metasprache der Schemasprache legt bereits eine gewisse Strukturierung aller Schemata zugrunde, da jedes gültige DCD-Schema definitionsgemäß ein RDF-Dokument darstellt. Ebenso ist die Semantik der eingesetzten RDF-Elemente bereits durch diese Spezifikation vorgegeben. Beide Punkte zusammengekommen offenbaren eine ausgeprägte Abhängigkeit von den weiteren RDF-Aktivitäten des World Wide Web Consortiums, die bisher nicht auf die Interdependenz von Schemasprache und Wissensbeschreibungsformat ausgerichtet ist. Positiv fällt an DCD die Verwendung von XML zur Beschreibung von XML-Sprachen auf, womit auch die letzte der erhobenen Anforderungen zu erfüllen wäre.

Die Verknüpfung von RDF mit DCD als Schemasprache birgt allerdings ein potientiell Problem hinsichtlich der Validierbarkeit der entstehenden Strukturen. Durch den Rückgriff von DCD auf RDF entsteht bei der Angabe eines Schemas für RDF ein transitiver Zirkelschluß. In der Konsequenz wird zur Validierung eines XML-Dokuments, welches einer mittels DCD-formulierten Grammatik

folgt, neben dem eigentlichen DCD-Schema des Dokuments auch das DCD-Metaschema und dessen Semantik-liefernde RDF-Beschreibung benötigt.

Diese Beschränkung mildert die vierte Familie von XML-Schemasprachen ab. Sie umfaßt die meisten Vorschläge, die alle als eigenständige XML-Sprachen ausgelegt sind; daher definieren sie ein eigenständiges XML-Vokabular zur Darstellung der benötigten XML-Strukturen, sowie die zugehörige Semantik.

In der Folge sind sie für die Meta-Schemaebene selbstbeschreibend. Das bedeutet das Schema eines Schemas kann durch sich selbst validiert werden. Da dieser Validierungsschritt statisch nur einmal erfolgen muß, kann er durch Schemawerkzeuge vorweggenommen werden.

In dieser Kategorie sind die meisten der bisher vorgeschlagenen Schemadialekte einzuordnen.

Die größte Bedeutung haben kontextfreie reguläre Sprachen zur Spezifikation von XML-Sprachstrukturen erlangt.

Eine Sprache dieses Typs entwickelt auch die [W3C-Arbeitsgruppe zur Definition eines XML-Schemasprachstandards](#). Insbesondere berücksichtigt diese Aktivität explizit die Vorgängersprachen [XML Data](#), [DCD](#), [SOX](#) sowie [Document Definition Markup Language](#). Die erwähnten konkurrierenden Vorschläge unterscheiden sich semantisch lediglich in Nuancen, bieten dem Anwender jedoch teilweise (optisch) stark unterschiedliche Konstrukte zur Syntaxspezifikation an.

Einen strukturell unterschiedlichen Ansatz verfolgt die durch Rick Jelliffe vorgeschlagene Sprache [Schematron](#). Sie interpretiert ein Schema als Sammlung von Regeln, denen ein gegebenes Dokument genügen muß, um als gültig akzeptiert zu werden. Dies erlaubt die Formulierung mächtiger kontextsensitiver Einschränkungen, die während des Validierungsvorganges geprüft werden.

Die Umsetzung dieser Schemasprache setzt auf den XML-Standards [XPath](#) und [XSLT](#) auf.

W3Cs XML-Schema:

Jenseits aller existierenden verschiedenen Sprachvorschläge kommt dem W3C-Standard der *XML Schema Description Language* (XSD) die größte praktische Bedeutung zu.

Tim Berners-Lee verkündete in der Eröffnungsrede der WWW-Konferenz in Hong Kong am 2. Mai 2001 die Verabschiedung als Recommendation. Gleichzeitig deutete er bereits weitere Schema-Aktivitäten des World Wide Web Consortiums an.

XML-Schema bildet zusammen mit XML v1.0 2nd edition und den Namensräumen die Basis aller weiteren W3C-XML-Sprachstandards.

Aus formalen Gründen ist nicht mit dem Ersatz der DTD durch Schema zu rechnen. Jedoch werden mittelfristig neu entwickelte XML-Sprachen keine Grammatiken mehr in der Syntax der DTD entwickeln, sondern direkt Schemata definieren.

XSD bildet eine vollständig in XML-Syntax formulierte kontextfreie reguläre Grammatik zur Formulierung beliebiger XML-Strukturen ab. Hierbei handelt es sich um die bekannten Grundprimitive [Element](#) und [Attribut](#)

Gleichzeitig wurde, neben zahlreichen anderen Neuerungen, die Kommentarsyntax für Schemata neu definiert.

Inhaltlich gliedert sich der XSD-Sprachvorschlag in zwei große Teilbereiche: [Part 1: Structures](#) zur Definition von Inhaltsmodellen für Elemente, Attributstrukturen und wiederverwendbaren Strukturen und [Part 2: Datatypes](#) zur Festlegung diverser inhaltlicher Charakteristika wie Datentypen und konsistenzgarantierende Einschränkungen.

In beiden Teilen werden [XML-Namensräume](#) explizit berücksichtigt. Konzeptionell rekonstruiert XSD-Part1 zunächst die bekannte Mächtigkeit der DTD um so die evolutionäre Weiterentwicklung bestehender XML-Sprachen zu ermöglichen.

Der zweite Teil der XSD-Spezifikation definiert ein eigenständiges Typsystem, das neben der naheliegenden Verwendung im ersten Teil der Schemasprache XSD auch in anderen W3C-Arbeitsgruppen Verwendung findet. Inhaltlich baut auch Part2 auf den in der DTD definierten Typen auf und erlaubt zunächst direkt ihre Angabe in Schemata. Darauf aufbauend wird eine Fülle verschiedenster Typen angeboten, die an die verschiedenen verfügbaren Typsysteme aus den Programmiersprachen, Datenbanken und internationalen Standards angelehnt sind.

Alle durch XSD definierten Elemente, d.h. alle Primitive zur Definition eines eigenen Schemas, befinden sich im Namensraum <http://www.w3.org/2001/XMLSchema>, der üblicherweise an das Präfix `xsd` gebunden wird. Elemente und Attribute aus XML-Schema, die in Instanzdokumenten verwendet werden können sind im Namensraum <http://www.w3.org/2001/XMLSchema-instance> (übliches Präfix `xsi`) organisiert.

Wegen des Umfangs der offiziellen Schemadokumente wird zusätzlich durch das W3C ein [Part 0: Primer](#) herausgegeben. Er stellt die beiden XSD-Teile in der Zusammenschau an Beispielen dar.

Schemareferenz:

Jedes XML-Schema bildet als XML-Dokument eine eigenständige Speichereinheit, üblicherweise eine Datei.

Die Verbindung zwischen Schema und beschriebenem Dokument wird durch das in der XSD-Spezifikation vordefinierte Attribut [schemaLocation](#) bzw. [noNamespaceSchemaLocation](#) definiert. Eines dieser Attribute muß zwingend im Wurzelement des XML-Dokuments angegeben werden. Legt das Schema keinen Namensraum für die enthaltenen Deklarationen fest, d.h. alle darin deklarierten Elemente befinden sich im Vorgabennamensraum, so findet sich die Schemareferenz in `noNamespaceSchemaLocation`; andernfalls in `schemaLocation`.

Das nachfolgende Beispiel zeigt die Deklaration:

Beispiel 20: Definition einer Schemareferenz


```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<ProjektVerwaltung
(3)    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(4)    xsi:schemaLocation="http://www.heckle.de/vorlesung/xml/examples/
projektverwaltung.xsd">
(5)    ...
```

Im Beispiel wird zunächst der XML-Schema-Instanzen-Namensraum an das Präfix `xsi` gebunden. Dies ermöglicht die Einbindung von Elementen und Attributen aus der Schemaspezifikation in das eigene Dokument.

Als erste Nutzung eines solchen Elements aus XSD wird das Attribut `schemaLocation` im Wurzelement mit der URI des Schemas als Wert belegt. Die Deklaration des XSI-Namensraumes ist daher zwingend. Die angegebene URI kann zur Ermittlung des Schemas für Validierungszwecke durch einen XML-Prozessor genutzt werden.

Aufbauend auf dem Begriff der [Wohlgeformtheit](#) definiert XML-Schema den der *Schemagültigkeit* als höhere Qualitätsstufe eines XML-Vokabulars:

**Definition 10: Gültigkeit hinsichtlich eines Schemas**

Ein XML-Dokument heißt *gültig hinsichtlich eines Schemas* (*schema valid*), wenn es über ein Schema verfügt, und konform zu diesem aufgebaut ist.

Aufgrund der Realisierung der Schemasprache als XML-Sprache ist jedes Schema auch ein XML-Dokument. Daher eröffnet sich die Möglichkeit, das Schema selbst durch ein Schema zu beschreiben. Dieses *Schema für Schema* -- auch *Metaschema* genannte -- XML-Dokument erlaubt die Validierung (im Sinne der *schema validness*) jedes Schemas. Damit erfüllt sich eine der Anforderungen an den Schemamechanismus: die Validierbarkeit der erstellten Schemata selbst, was für DTDs nicht gegeben war. In der praktischen Anwendung zeigt sich dies in der Möglichkeit, erstellte Schemata mit denselben Werkzeugen zu analysieren, verarbeiten und zu prüfen, die auch für Instanzdokumente verwendet werden.

Da das Metaschema selbst wiederum ein XML-Dokument ist, folgt, daß hierfür auch ein Schema angegeben werden kann. Die XML-Standardisierung hat hier -- nicht zuletzt um eine unendliche Reihung zur Validierung notwendiger Schemata zu vermeiden -- den Ansatz gewählt, das Schema für Schema durch sich selbst zu beschreiben.

Die Abbildung stellt die getroffenen Aussagen und Validierungsbeziehungen nochmals graphisch zusammen.



erkennbar, dem nicht explizit unterstützten Zugriff auf die Komponenten ([date](#) bietet keine Zugriffsmöglichkeit auf die Komponenten Tag, Monat und Jahr), verweisen.

Die aggregierten Typen teilen sich in Listen-artige und Vereinigungstypen (*Union*). Listen bestehen dabei aus einer geordneten Menge atomarer Typen.

Vereinigungstypen erlauben hingegen die Verknüpfung Typ-verschiedener Datentypen zu einem neuen.

- [Primitive und abgeleitete Typen](#)

Primitive Datentypen existieren unabhängig von anderen Datentypen, während abgeleitete Datentypen von der Definition ihres Elterntyps abhängig sind.

- [Vorgegebene und anwenderdefinierte Typen](#)

Die vorgegebenen Typen sind diejenigen, die in [XML-Schema Part2](#) beschrieben sind. Alle weiteren Typen eines Schemas sind durch den Anwender definierte abgeleitete Typen.

Durch Erweiterungs- und Aggregationsmechanismen ergibt sich das in der nachfolgenden Abbildung dargestellte Typsystem.

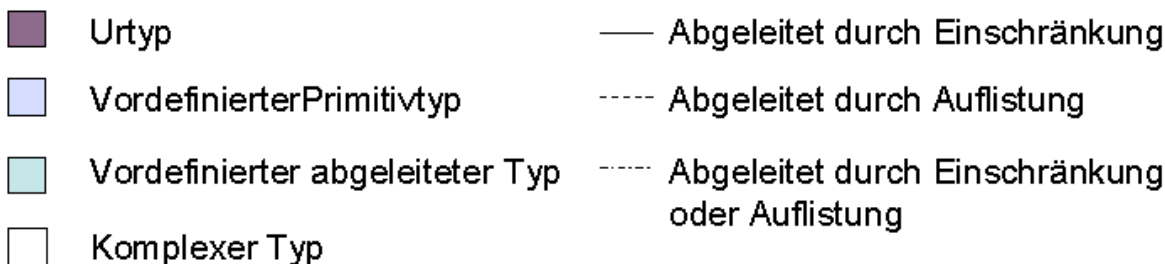
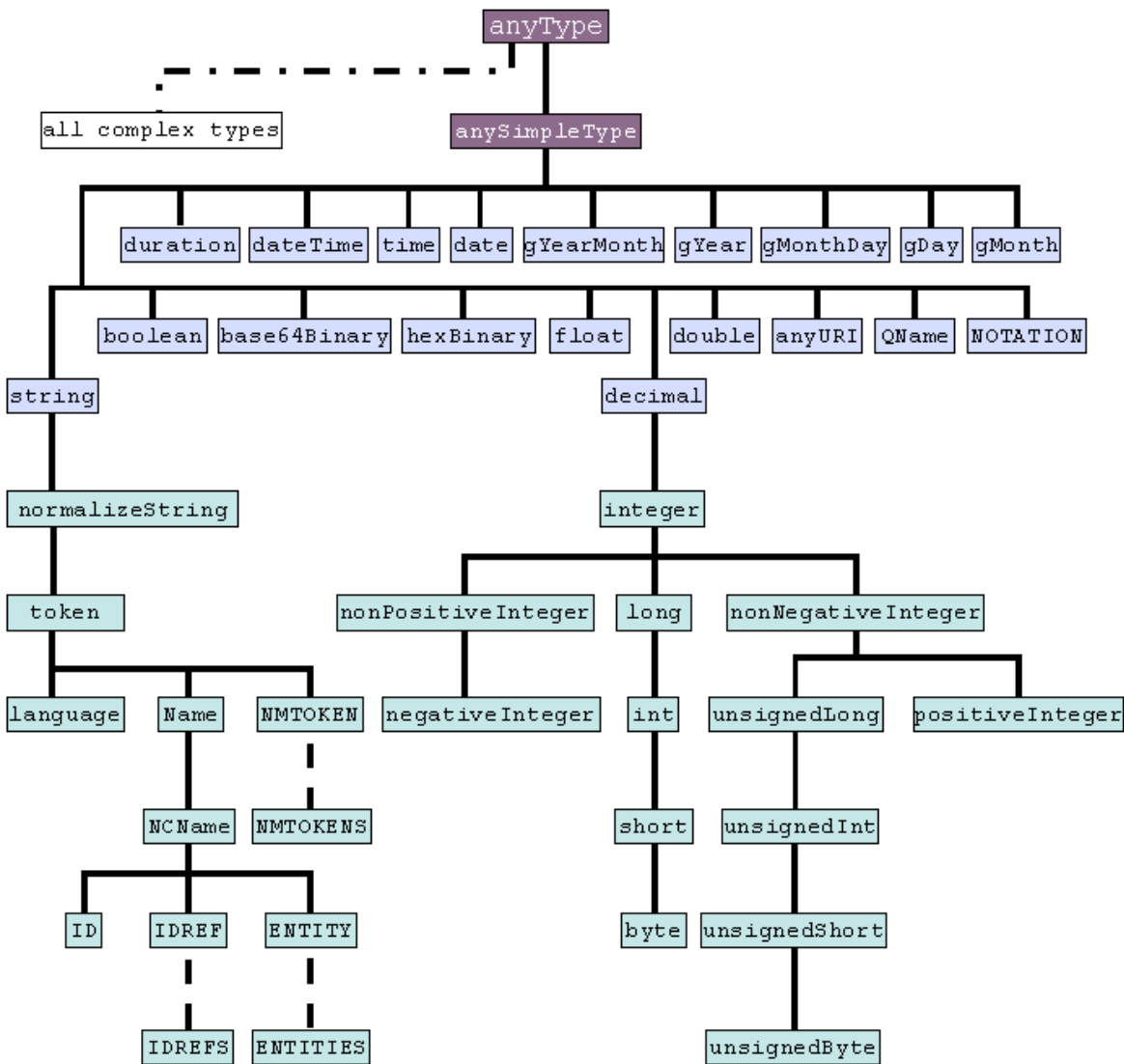


Abbildung 15: Das XSD-Typsensystem (click on image to enlarge!)

Die Tabelle stellt die angebotenen Typen mit einigen Beispielen dar:

Tabelle 6: Typen in XSD-Schema Part 2

Typname	Beispiel	Bemerkung
string	Hello 
 World	Jedes beliebige Unicode Symbol gemäß XML-Syntaxproduktion 2
normalizedString	&#20Hello&#20;World	Jedes beliebige Unicode Symbol außer Zeilenvorschub, Wagenrücklauf und Tabulatoren normalizedString ist eine einschränkende Spezialisierung des Typs string
token	Hello World	Jeder normalizedString , unter Weglassung führender, abschließender und mehrfacher Leerzeichen (<code>#x20</code>), sowie Zeilenvorschüben (<code>#xA</code>) und Tabulatoren (<code>#x9</code>). token ist eine einschränkende Spezialisierung des Type normalizedString
Name	aName, _helloWorld, : notAGoodIdea	XML Name gemäß Syntaxproduktion 5 . Name ist eine einschränkende Spezialisierung des Typs token
QName	xsd:element, element	Durch Namensraumpräfix qualifizierter Name gemäß Produktion 6 der XML Namespace Recommendation
NCName	aName, _anotherName, X	Name, der keinen Doppelpunkt enthält (<i>non colonized name</i>), gemäß Produktion 4 der XML Namespace Recommendation
decimal	-1.23, 12678967.543233, +100000.00, 210	Wertebereich: $i \cdot 10^{-n}$, mit i , n aus integer , $n \geq 0$ Ein Prozessor muß mindestens 18 Dezimalstellen unterstützen
long	-9223372036854775808, ... -1, 0, 1, ... 9223372036854775807	Wertebereich: $2^{63} \leq \text{long} \leq 2^{63}-1$ long ist eine einschränkende Spezialisierung des Typs integer
int	-2147483648, ... -1, 0, 1, ... 2147483647	Wertebereich: $-2^{31} \leq \text{int} \leq 2^{31}-1$ int ist eine einschränkende Spezialisierung des Typs long
short	-32768, ... -1, 0, 1, ... 32767	Wertebereich: $-2^{15} \leq \text{short} \leq 2^{15}-1$ short ist eine einschränkende Spezialisierung des Typs int
byte	-128, ...-1, 0, 1, ... 127	Wertebereich: $-2^7 \leq \text{byte} \leq 2^7-1$ byte ist eine einschränkende Spezialisierung des Typs short
integer	...-1, 0, 1, ...	Wertebereich: entspricht der mathematischen Menge der ganzen Zahlen ($\mathbb{Z}$) integer ist eine einschränkende Spezialisierung des Typs decimal

positiveInteger	1, 2, ...	Wertebereich: entspricht der mathematischen Menge der natürlichen Zahlen (N) positiveInteger ist eine einschränkende Spezialisierung des Typs nonNegativeInteger
negativeInteger	... -2, -1	Wertebereich: {..., -2, -1}, die unendliche Menge der negativen Zahlen negativeInteger ist eine einschränkende Spezialisierung des Typs nonPositiveInteger
nonNegativeInteger	0, 1, 2, ...	Wertebereich: $0 \leq$ nonNegativeInteger nonNegativeInteger ist eine einschränkende Spezialisierung des Typs integer
nonPositiveInteger	... -2, -1, 0	Wertebereich: {..., -2, -1, 0} die unendliche Menge der negativen Zahlen, und die Null nonPositiveInteger ist eine einschränkende Spezialisierung des Typs integer
unsignedLong	0, 1, ... 18446744073709551615	Wertebereich: $0 \leq$ unsignedLong $\leq 2^{64}-1$ unsignedLong ist eine einschränkende Spezialisierung des Typs nonNegativeInteger
unsignedInt	0, 1, ... 4294967295	Wertebereich: $0 \leq$ unsignedInt $\leq 2^{32}-1$ unsignedInt ist eine einschränkende Spezialisierung des Typs unsignedLong
unsignedShort	0, 1, ... 65535	Wertebereich: $0 \leq$ unsignedShort $\leq 2^{16}-1$ unsignedShort ist eine einschränkende Spezialisierung des Typs unsignedInt
unsignedByte	0, 1, ... 255	Wertebereich: $0 \leq$ unsignedByte $\leq 2^8-1$ unsignedByte ist eine einschränkende Spezialisierung des Typs unsignedShort
float	-1E4, 1267.43233E12, 12.78e-2, 12, INF	32-Bit-Zahl mit einfacher Genauigkeit gemäß IEEE 754-1985 . Wertebereich: $m * 2^e$, wobei m und integer -Elemente mit $m \leq 2^{24}$, und $-149 \leq e < 104$ sind.
 double	-1E4, 1267.43233E12, 12.78e-2, 12, INF	64-Bit-Zahl mit doppelter Genauigkeit gemäß IEEE 754-1985 . Wertebereich: $m * 2^e$, wobei m und integer -Elemente mit $m \leq 2^{53}$, und $-1075 \leq e < 970$ sind.
boolean	true, false, 1, 0	Unterstützung der klassischen zweiwertigen Logik
time	13:20:00-05:00, 13:20:00.000	Uhrzeit, die täglich wiederkehrt, ausgedrückt im Format gemäß ISO 8601
date	2004-01-08	Datumsformat: CCYY-MM-DD, gemäß ISO 8601

gYear	1999, 2001, 2004	Darstellung von Jahren des gregorianischen Kalenders gemäß ISO 8601
gYearMonth	2004-01	Darstellung eines Monats eines bestimmten Jahres des gregorianischen Kalenders gemäß ISO 8601
gDay	----05, ----31	Darstellung eines wiederkehrenden Tages eines Monats gemäß ISO 8601
gMonthDay	--31-12, --01-01	Darstellung eines wiederkehrenden gregorianischen Datums, gebildet aus Tag Monat und Monat im Format --MM-DD, gemäß ISO 8601
gMonth	--03, --12	Monatsformat: --MM-- gemäß ISO 8601
dateTime	2004-01-08T22:57:06.000+02:00	Zeitpunkt, ausgedrückt durch Datum und Uhrzeit; beide gemäß ISO 8601 codiert.
duration	P1Y2M3DT10H30M12.3S Zeitraum von einem Jahr, zwei Monaten, drei Tagen, zehn Stunden, 30 Minuten und 12,3 Sekunden	Nach Größe (Signifikanz) geordnete Koordinate im sechs-dimensionalen Raum aus Jahr, Monat, Tag, Stunde, Minute und Sekunde. Formatdefinition laut ISO 8601
base64Binary	SGVsbG8gd29ybGQhCg==	Base64-Darstellung eines beliebigen Binärinterpretierten Inhaltes gemäß IETF RFC 2045
hexBinary	0FB7	Hexadezimale Darstellung beliebiger Binärinterpretierter Inhalte
anyURI	http://www.jeckle.de	Jede gemäß IETF RFC 2396 bzw. IETF RFC 2732 gültige URI
language	en-GB, en, de-de	Sprachcodierung gemäß IETF RFC 1766 und XML Recommendation language identification . Die Identifikationsnamen werden durch ISO 639 sowie ISO 3166 definiert. language ist eine einschränkende Spezialisierung des Typs token
ID	test, XYZ	XSD-Darstellung des DTD-Typen ID . Zugelassen sind alle Ausprägungen der Namespaceproduktion 4 (NCName) . ID ist eine einschränkende Spezialisierung des Typs NCName
IDREF	test, XYZ	XSD-Darstellung des DTD-Typen IDREF . Zugelassen sind alle Ausprägungen der Namespaceproduktion 4 (NCName) . IDREF ist eine einschränkende Spezialisierung des Typs NCName

IDREFS	test1 test2 test4, test3 test5	XSD-Darstellung des DTD-Typen IDREFS . Zugelassen sind Listen aus white space separierten Ausprägungen der Namespaceproduktion 4 (NCName) . IDREFS ist eine nichtleere Aufzählung von IDREF -Ausprägungen
ENTITY		XSD-Darstellung des DTD-Typen ENTITY. Zugelassen sind alle Satzformen, die der Produktion NCName der XML-Namensräume entsprechen und als ungeparste Entität definiert sind. ENTITY ist eine einschränkende Spezialisierung des Typs NCName
ENTITIES		XSD-Darstellung des DTD-Typen ENTITIES. Zugelassen sind Listen aus white space separierten Ausprägungen des Typs ENTITY . ENTITIES ist eine nichtleere Aufzählung von ENTITY -Ausprägungen
NOTATION		XSD-Darstellung des DTD-Typen NOTATION. Zur Verwendung dieses Typs in einem Schema muß eine Ableitung von NOTATION durch den Anwender definiert werden.
NMTOKEN	US, Deutschland	XSD-Darstellung des DTD-Typen NMTOKEN. Ausprägungen dieses Typs müssen konform zur Produktion 7 der XML-Spezifikation sein. NMTOKEN ist eine einschränkende Spezialisierung des Typs token
NMTOKENS	US UK Aus, Ger	XSD-Darstellung des DTD-Typen NMTOKENS. Zugelassen sind Listen aus white space separierten Ausprägungen des Typs NMTOKEN . NMTOKENS ist eine nichtleere Aufzählung von NMTOKEN -Ausprägungen
anyType	1, 2.3, aGVsb, 06b8f45, test&#20;für&#20;anyType&#0A;<sentence>the quick brown <animal>fox</animal>...</sentence>	Allgemeinster Datentyp. Konzeptionell bildet er die Vereinigung aller angebotenen XSD-Typen.

Die einfachste Form zur **Definition eines Elements mit unstrukturiertem typisierten Inhalt** lautet:

```
<xsd:element
    name="elementName"
    type="typeName"/>
```

XSD definiert ferner folgende Charakteristika für Elemente, die durch Attribute der Elementdeklaration ausgedrückt werden:

- **abstract**: falls auf true gesetzt, darf ein solches Element nicht in einem XML-Dokument auftreten. Es kann ausschließlich zur Strukturierung des Schemaentwurfs eingesetzt werden und als Basis von Spezialisierungen dienen. Vorgabewert ist false
- **block**: erlaubt die Kontrolle der Verwendung abgeleiteter Typen. Zugelassene Belegungen beliebige Kombinationen aus extension, restriction und substitution oder der Einzelwert #all. Ist das Attribut auf restriction gesetzt, so dürfen keine (einschränkend) abgeleiteten Typen Ausprägungen des Originaltyps in Instanzdokumenten ersetzen. Dasselbe gilt für extension oder als Substitutionsgruppen deklarierte Typen (substitution-Belegung). Der Wert #all versammelt alle möglichen Varianten und verbietet generell die Ersetzung eines Elements durch andere.
- **default**: Vorgabebelegung des Inhalts durch eine beliebige Zeichenkette, die konform zum gewählten Typ ist. Dieser Wert wird durch den XML-Prozessor an die Applikation gemeldet, wenn kein Wert im Dokument angegeben wird.
- **final**: verhindert die Ableitung von Typen. Die zulässigen Belegungen sind mit denen für block identisch, nur daß durch dieses Attribut die Vererbungsmechanismen bereits auf Schemaebene verboten werden, während block ihre Nutzung im Instanzdokument einschränkt.
- **fixed**: erlaubt die konstante Wertbelegung.
- **form**: legt fest, ob das Element im Instanzdokument mit Namensraumpräfix erscheint. Zulässige Belegungen: qualified (Namensraumpräfix muß angegeben werden) und unqualified.
- **id**: erlaubt die eindeutige Kennzeichnung eines Elements durch eine Schema-weit eindeutige Zeichenkette.
- **minOccurs**: Minimalkardinalität, d.h. Mindestzahl zulässiger Vorkommen dieses Elements. Der Attributinhalt ist ein Element aus [nonNegativeInteger](#). Das Attribut ist optional, und wird bei fehlender Angabe mit dem Vorgabewert 1 belegt.
- **maxOccurs**: Maximalkardinalität, d.h. Höchstzahl zulässiger Vorkommen dieses Elements. Der Attributinhalt ist entweder ein Element aus [nonNegativeInteger](#) oder die Zeichenkette unbounded zur Kennzeichnung beliebig vieler Auftreten. Das Attribut ist optional, und wird bei fehlender Angabe mit dem Vorgabewert 1 belegt.
- **name**: Unqualifizierter Name des Elements, konform zur [NCName](#)-Produktion der Namensraum-Spezifikation.
- **nillable**: Erlaubt Null-Werte im Instanzdokument, die Semantik ist dabei an die in relationalen Datenbanksystemen verwirklichte angelehnt. Die Belegung ist entweder true oder false, was auch als Vorgabe bei Fehlen dieses Attributs angenommen wird.
- **ref**: Referenz auf eine andere Elementdeklaration zur Übernahme der dort spezifizierten Definitionen.
- **substitutionGroup**: Name einer Gruppe von Elementen, die anstatt des aktuellen Elements im Instanzdokument auftreten dürfen.
- **type**: Ein durch [Schema Part 2 vordefinierter Typ](#), oder jeder beliebige anwenderdefinierte.

Nachfolgend sind einige Elementdeklarationen für unstrukturierten Inhalt versammelt

Beispiel 21:

```
(1)<element name="geburtsdatum" type="xsd:date"/>
(2)<element name="pi"
(3)    type="xsd:double"
(4)    fixed="3.141592653"
(5)    block="#all"
(6)    final="#all"/>
(7)<element name="vorname"
(8)    type="xsd:token"
(9)    minOccurs="1"
(10)   maxOccurs="unbounded"/>
(11)<element name="artikelNummer"
(12)   type="xsd:NCName"
(13)   form="qualified"/>
```

Die Deklaration geburtsdatum definiert ein XML-Element des Typs [date](#) zur Darstellung eines Datums. Weitere Festlegungen sind nicht getroffen, daher wird das Element mit minOccurs und maxOccurs 1 belegt, wodurch es als zwingend anzugebend (*mandatory*) und skalar (d.h. nicht mengenwertig) ausgewiesen wird.

pi legt die gleichnamige mathematische Konstante fest. Als Datentyp wurde [double](#), eine Gleitkommazahl mit doppelter Genauigkeit gewählt. Als konstante Belegung wird durch das fixed Attribut der entsprechende Zahlenwert festgelegt. Daher muß eine Vorgabebelegung durch das Attribut default nicht erfolgen; gemäß Schema-Spezifikation darf sie sogar nicht erfolgen, fixed

und default schließen sich gegenseitig aus. Um eine weitere Spezialisierung des Elements durch Vererbung oder Aggregation zu verhindern wird der Wert von block auf #all gesetzt, wodurch die Teilnahme an allen Typbildungsmechanismen unterbunden wird.

Die Definition für vorname nutzt als Datentyp den [token](#), der automatisch mehrfache, führende und abschließende Leerzeichen sowie sonstige Formatierungssymbole entfernt. Ferner kann dieses Element beliebig häufig auftreten -- maxOccurs ist daher auf unbounded gesetzt. Die Fixierung der minimalen Auftrittshäufigkeit auf 1 (minOccurs) entspricht der Vorgabebelegung.

Für das Element artikelNummer ist als Typ NCName ausgewählt, was beliebigen Zeichenketten -- die keinen Doppelpunkt enthalten -- entspricht. Darüberhinaus ist das Attribut form mit dem Wert qualified versehen. Dies führt dazu, daß das Namensraumkürzel für dieses Element zwingend im Instanzdokument anzugeben ist.

Zur Umsetzung des freien Inhaltsmodells, das beliebige Inhalte aus den definierten Elementen und freien Texten zuläßt, wird ebenfalls auf das Typsystem zurückgegriffen.

Wird das type Attribut nicht belegt, so wird gemäß Vorgabe der Typ [anyType](#) angenommen.

Elemente dieses Typs können beliebige [wohlgeformte](#) Inhalte beherbergen.

Die beiden nachfolgenden Angaben sind daher äquivalent.

```
<element
  name="elementName"
  type="xsd:anyType"/>
<element name="elementName"/>
```

XSD prägt den bereits im Kontext der DTD genutzten Typbegriff (dort beschränkt er sich lediglich auf verschiedene Darstellungsformen uninterpretierter Zeichenketten) strenger. Dies zeigt sich deutlich in der Existenz des XSD-Elements [complexType](#). Es führt die Möglichkeit einer expliziten, d. h. von der Verwendung losgelösten Typbildung, ein. Syntaktisch kann die complexType-Definition sowohl innerhalb einer Elementdefinition, als auch separat erfolgen.

Den einfachsten Anwendungsfall bildet die eingebettete leere complexType-Definition zur Darstellung des **leeren Inhaltsmodells**.

Die Syntax hierfür lautet (der XSD-Namensraum sei an das Präfix xsd gebunden):

```
<xsd:element
  name="elementName">
  <xsd:complexType/>
</xsd:element>
```

Ein XML-Schema-validierender Parser verhält sich in diesem Falle identisch zu einem (DTD-) validierenden Parser. Daher werden für die obige Festlegung ausschließlich die beiden Darstellungsformen zur Angabe eines leeren Elements (<elementName/> bzw. <elementName></elementName>) akzeptiert.

Die Befüllung des complexType-Elements leitet direkt zum wichtigsten Inhaltsmodell über, dem explizit angegebener Kindelemente.

Zur Festlegung der Elementreihenfolge definiert XML-Schema das Element sequence, welches die Angabe der Kindelemente in genau der im Schema angegebenen Reihenfolge erzwingt.

Das Auswahlinhaltsmodell (auch: Selektionsmodell) --- welches alternativ das Auftreten beliebiger Elemente definiert --- wird entsprechend durch das XSD-Element [choice](#) ausgedrückt.

Eine besondere Variante des Selektionsmodells stellt die [all](#)-Gruppe dar. Es erlaubt die Angabe der Kindelemente in beliebiger Reihenfolge.

Die drei Ausgangsvarianten können im Rahmen einer Elementdefinition beliebig geschachtelt und auf diesem Wege kombiniert werden.

Am Beispiel der [Elementdefinitionen der Projektverwaltung](#):

Beispiel 22: Einige Elementdefinitionen

```
(1)<?xml version = "1.0" encoding = "UTF-8"?>
(2)<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
(3)    <xsd:element name = "Projektverwaltung">
(4)        <xsd:complexType>
(5)            <xsd:sequence>
(6)                <xsd:element ref = "Person" maxOccurs = "unbounded"/>
(7)                <xsd:element ref = "Projekt" maxOccurs = "unbounded"/>
(8)            </xsd:sequence>
(9)        </xsd:complexType>
(10)    </xsd:element>
(11)    <xsd:element name = "Person">
(12)        <xsd:complexType>
(13)            <xsd:sequence>
(14)                <xsd:element name = "Vorname" type = "xsd:token"
```



```

maxOccurs = "unbounded"/>
(15)                                     <xsd:element name = "Nachname" type = "xsd:token"/>
(16)                                     <xsd:element ref = "Qualifikationsprofil" minOccurs
= "0"/>
(17)                                     </xsd:sequence>
(18)                                     </xsd:complexType>
(19)     </xsd:element>
(20)     <xsd:element name = "Projekt">
(21)         <xsd:complexType/>
(22)     </xsd:element>
(23)     <xsd:element name = "Qualifikationsprofil">
(24)         <xsd:complexType mixed = "true">
(25)             <xsd:sequence>
(26)                 <xsd:element name = "Qualifikation" type = "xsd:
string" minOccurs = "0" maxOccurs = "unbounded"/>
(27)                 <xsd:element name = "Leistungsstufe" type = "xsd:
string" minOccurs = "0" maxOccurs = "unbounded"/>
(28)             </xsd:sequence>
(29)         </xsd:complexType>
(30)     </xsd:element>
(31)</xsd:schema>

```

Download des Beispiels

Das Schema enthält alle Elementdefinitionen für die Projektverwaltung. Innerhalb jedes element-Elements sind die entsprechenden Kindelemente in sequence-Strukturen eingebettet. Die Elemente müssen daher in der Reihenfolge ihres Auftretens im Schema auch im Instanzdokument wiedergegeben werden.

Von besonderem Interesse ist die Definition des *Qualifikationsprofils*. Es handelt sich dabei um ein **mixed content model**, ausgedrückt durch das Boole'sche Attribut *mixed* ([in Spezifikation nachschlagen](#)).

Darüberhinaus enthält das Beispiel neben *lokalen Elementdeklarationen*, die sich vollständig im Elternelement finden (wie Vorname, Nachname und Qualifikation), auch *globale Elementdeklarationen*, die zunächst deklariert und in einem zweiten Schritt durch Referenzierung als Kindelemente verwendet werden (wie Person und Projekt innerhalb Projektverwaltung, oder Qualifikationsprofil innerhalb des Elements Person). Hierdurch können vollständige Elemente an verschiedenen Stellen im Schema referenziert und so verwendet werden. Die Definition ist der lokalen ebenbürtig und wird im Instanzdokument identisch behandelt. Zusammenfassend läßt sich festhalten: Mit dem Referenzierungsmechanismus für Elemente kann eine einfache Form der Wiederverwendung umgesetzt werden.

Den Zeichenketten-artigen Elementtypen wurde durchgehend der XSD-Typ [string](#) zugewiesen.

Durch die Referenzierungsmöglichkeit existiert eine erste Möglichkeit zur Wiederverwendung bereits im Schema definierter Elemente. Jedoch werden Elemente hierbei zwingend in ihrer vollständigen Definition, d.h. Name, Typ und Inhaltsmodell, eingebunden.

XML-Schema bietet die Möglichkeit, strukturierte Typen, die ausschließlich durch ihr Inhaltsmodell definiert werden, festzulegen. In der Konsequenz verändert sich der durch die DTD formulierte Typbegriff hin zu einer eher an den Programmiersprachen orientierten Sichtweise, da die Benennung des Typs von der Namensgebung der typisierten Instanz separiert wird. Syntaktisch erfolgt die Typbildung durch die Benennung des *complexType*-Elements durch ein Attribut *name*. Um die mehrfache Verwendung eines solchen Typen zu ermöglichen, muß seine Definition zwingend auf einer Baumstufe erfolgen, die für alle nutzenden Elemente erreichbar ist. Üblicherweise werden daher diese Definitionen auf der ersten Stufe, direkt unterhalb des Wurzelknotens, platziert.

Zur Unterscheidung dieser *benannten komplexen Typen* werden die bisher genutzten -- namenlosen Typen -- als *anonyme komplexe Typen* bezeichnet.

Das nachfolgende Beispiel zeigt die Definition eines benannten komplexen Typen am Beispiel des Elements Person:

Beispiel 23: Nutzung benannter komplexer Typen



```

(1)<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
(2)  <xsd:complexType name="PersonType">
(3)    <xsd:sequence>
(4)      <xsd:element name = "Vorname" type = "xsd:string"
(5)        maxOccurs = "unbounded"/>
(6)      <xsd:element name = "Nachname" type = "xsd:string"/>
(7)      <xsd:element ref = "Qualifikationsprofil" minOccurs = "0"/>
(8)    </xsd:sequence>
(9)  </xsd:complexType>
(10)
(11)  <xsd:element name = "ProjektVerwaltung">
(12)    <xsd:complexType>
(13)      <xsd:sequence>
(14)        <xsd:element name="Person" type="PersonType" maxOccurs = "unbounded"/>
(15)        <xsd:element ref = "Projekt" maxOccurs = "unbounded"/>
(16)      </xsd:sequence>
(17)    </xsd:complexType>
(18)  </xsd:element>
(19)
(20)  <xsd:element name = "Projekt">
(21)    <xsd:complexType/>
(22)  </xsd:element>
(23)
(24)  <xsd:element name = "Qualifikationsprofil">
(25)    <xsd:complexType mixed = "true">
(26)      <xsd:sequence>
(27)        <xsd:element name = "Qualifikation" type = "xsd:string"
(28)          minOccurs = "0" maxOccurs = "unbounded"/>
(29)        <xsd:element name = "Leistungsstufe" type = "xsd:string"
(30)          minOccurs = "0" maxOccurs = "unbounded"/>
(31)      </xsd:sequence>
(32)    </xsd:complexType>
(33)  </xsd:element>
(34)</xsd:schema>

```

[Download des Beispiels](#)

Das Schema zeigt die Definition des komplexen Typen PersonType. Dieser Typ wird zur Festlegung des Inhaltsmodells des Elements Person verwendet.

Definition eigener Datentypen durch Vererbung:

Zur Unterstützung von Wiederverwendung und Erhöhung der Strukturierung des Entwurfs definiert XSD ein Vererbungsstruktur zur Bildung neuer komplexer Typen auf der Basis bereits bestehender.

Zwei verschiedene Ableitungssemantiken werden angeboten:

- Ableitung durch Einschränkung (*derivation by restriction*)
Der ererbende Subtyp gibt eine engere Definition des Supertypen
- Ableitung durch Erweiterung (*derivation by extension*)
Der ererbende Subtyp erweitert die Definition des Supertypen

Das nachfolgende Beispiel zeigt die Anwendung der einschränkenden Ableitung.

Hierbei erbt der benannte komplexe Typ childType von parentType. Innerhalb des -- aus syntaktischen Gründen notwendigen -- Elements complexContent findet sich die Definition der Vererbung im Element restriction, das base-Attribut verweist auf den benannten Elterntypen. Der Inhalt des restriction-Elements gleicht der Inhaltsmodelldefinition des komplexen Typen: Auch hier werden Elemente und ihre Auftretensstruktur (im betrachteten Beispiel sequence) angegeben. Die Elementdefinition des Elements elementA in childType schränkt die gleichnamige Elementdefinition innerhalb des Elterntypen ein. Nachvollziehbar wird diese Einschränkungsbziehung zwischen [short](#) und [int](#) bei Betrachtung der [Datentyphierarchie](#) und der Typdefinition der verwendeten Primitivtypen. So bildet short per definitionem eine eingeschränkte Untermenge von int an. (Die entsprechende [XSD-Definition](#) findet sich im *Schema für Schema*). Die beiden Elementdefinitionen usage1 und usage2 zeigen die Verwendung der anwenderdefinierten Typen.

Beispiel 24: Einschränkende Typableitung



```

(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
(3)  <xsd:complexType name="parentType">
(4)    <xsd:sequence>
(5)      <xsd:element name="elementA" type="xsd:int"/>
(6)    </xsd:sequence>
(7)  </xsd:complexType>
(8)
(9)  <xsd:complexType name="childType">
(10)    <xsd:complexContent>
(11)      <xsd:restriction base="parentType">
(12)        <xsd:sequence>
(13)          <xsd:element name="elementA" type="xsd:short"/>
(14)        </xsd:sequence>
(15)      </xsd:restriction>
(16)    </xsd:complexContent>
(17)  </xsd:complexType>
(18)
(19)  <xsd:element name="usage1" type="parentType"/>
(20)  <xsd:element name="usage2" type="childType"/>
(21)
(22)</xsd:schema>

```

Download des Beispiels

Durch das strukturierte Inhaltsmodell ergeben sich über die reine Typisierung hinausgehende Möglichkeiten zur Einschränkung der Inhalte. Die nachfolgende Tabelle stellt einige Varianten zusammen.

Tabelle 7: Beispiele für zulässige Restriktionen



Basistyp	Restriktion	Bemerkung
	default	Zusätzliche Belegung eines Elements mit einem Vorgabewert
	fixed	Beschränkung eines zunächst frei wählbaren Elements auf konstanten Inhalt
	type	Definition eines Typen für ein zunächst untypisiertes Element. (Auch hierbei handelt es sich um eine einschränkende Redefinition, da allen Elementen ohne Typdefinition standardmäßig der Typ anyType zugeordnet wird.)
minOccurs = n_1 , maxOccurs = m_1	minOccurs = n_2 , maxOccurs = m_2	Restriktion der Auftrittshäufigkeit auf eine geringere Anzahl. Daher gilt: $n_1 \leq n_2$ und $m_1 \geq m_2$

Die direkte Umkehrung der einschränkenden Spezialisierung bildet die *erweiternde Spezialisierung*. Sie greift nicht verändernd auf die Elemente des Supertyps zu, sondern definiert zusätzliche neue. Untenstehendes XSD-Schema zeigt dies am Beispiel des Supertyps `parentElement`, der durch das abgeleitete Kindelement `childElement` erweitert wird. Hierzu definiert `childElement` ein zusätzliches `elementB`.

Beispiel 25: Erweiternde Typableitung



```

(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
(3)  <xsd:complexType name="parentElement">
(4)    <xsd:sequence>
(5)      <xsd:element name="elementA"/>
(6)    </xsd:sequence>
(7)  </xsd:complexType>
(8)
(9)  <xsd:complexType name="childElement">
(10)    <xsd:complexContent>
(11)      <xsd:extension base="parentElement">

```

```

(12)                                <xsd:sequence>
(13)                                <xsd:element name="elementB"/>
(14)                                </xsd:sequence>
(15)                                </xsd:extension>
(16)                                </xsd:complexContent>
(17)    </xsd:complexType>
(18)</xsd:schema>

```

[Download des Beispiels](#)

Zusätzlich sieht XML Schema die Möglichkeit vor komplexe Typen von simplen abzuleiten. Dies mag auf den ersten Blick ungewöhnlich erscheinen, eröffnet es doch scheinbar einen Weg unstrukturierte Typen in strukturierte zu überführen.

Bei näherer Betrachtung offenbart sich jedoch, daß hier lediglich der Ableitungsbegriff überladen wurde um einen einfachen Weg zur Verknüpfung der beiden Inhaltsmodelle strukturierter „XML-artiger“ Inhalt -- wie er durch complexTypes repräsentiert wird -- auf der einen, und unstrukturiertem Inhalt -- wie er durch die [einfachen Datentypen](#) repräsentiert wird -- auf der anderen Seite.

Beispiel 26: Ableitung eines komplexen Typen von einem Simplen



```

(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xs:schema
(3)    xmlns:xs="http://www.w3.org/2001/XMLSchema"
(4)    elementFormDefault="qualified"
(5)    attributeFormDefault="unqualified">
(6)    <xs:element name="Vorname">
(7)        <xs:complexType>
(8)            <xs:simpleContent>
(9)                <xs:extension base="xs:string">
(10)                    <xs:attribute
(11)                        name="rufname"
(12)                        type="xs:boolean"/>
(13)                </xs:extension>
(14)            </xs:simpleContent>
(15)        </xs:complexType>
(16)    </xs:element>
(17)</xs:schema>

```

[Download des Beispiels](#)

Durch die im Beispiel dargestellte Syntax wird es ermöglicht unstrukturiert-getypten Elementen Attribute zuzuordnen, obwohl diese eigentlich Bestandteil der Definition komplex-getypter Elemente sind.

So wird im Beispiel dem Element Vorname sowohl der simple Typ string, als auch durch den Ableitungsmechanismus das Attribut rufname -- im Rahmen eines complexType, zugeordnet. Die Typisierung des Elements erfolgt hierbei nicht durch das type-Attribut innerhalb der Elementdeklaration, sondern innerhalb der simpleContent-Festlegung.

Neben der anwenderdefinierten Bildung komplexer Typen steht es dem XSD-Modellierer auch offen, eigene (primitive) Datentypen festzulegen oder eigene Typen von bestehenden abzuleiten. Hierfür definiert XML-Schema Part1 das Element [simpleType](#). Für einfache Typen ist jedoch nur die einschränkende Vererbung (*restriction*) zugelassen. Dies liegt in der praktischen Beherrschbarkeit des [Typsystems](#) begründet. Durch die strikte Restriktionssemantik ergibt sich die Möglichkeit kontravarianter Substitution, wie sie bei objektorientierten Typsystemen und Vererbungsstrukturen anzutreffen ist. Dies bedeutet, daß an jeder Stelle, an der eine Ausprägung eines Supertyps erwartet wird, auch -- unter Erhalt der Typrestriktion -- eine Ausprägung eines Subtypen auftreten darf. Beispielhaft: Wird an einer Stelle des Instanzdokumentes durch das Schema das Auftreten einer Ausprägung von [integer](#) verlangt, so kann der Anwender auch Ausprägungen der Subtypen [int](#), [short](#) oder [byte](#) angeben ohne die Gültigkeit des XML-Dokuments zu beeinträchtigen.

Vereinigungstypen werden aus einer nichtleeren Menge von Ausgangstypen gebildet. Das Beispiel zeigt die Definition eines Typen termin, der den vorgegebenen Primitivtypen [date](#) und eine Liste NamenDerWochentage (deren Definition nicht dargestellt ist) vereinigt. Insbesondere zeigt der Ausschnitt die Möglichkeit der Vereinigungsbildung auch über aggregierte Typen.

```

(1)<xs:simpleType name="termin">

```

```
(2)      <xs:union memberTypes="xs:date NamenDerWochentage"/>
(3)</xs:simpleType>
```

Das XSD-Beispiel zeigt, als Fragment der XML-Schemaspezifikation, die Definition des vorgegebenen Typs [short](#), einer einschränkenden Spezialisierung des Typs [int](#).

Am Beispiel gut nachvollziehbar sind die beiden Schritte zur Bildung eines eigenen Typen:

1. Auswahl eines Ausgangstypen (später Elementtyp (bei aggregierten Typen) oder Basistyp (bei abgeleiteten Typen))
2. Typdefinition durch Anwendung der entsprechenden Typkonstruktion und evtl. Einschränkung verschiedener Charakteristika

Im Beispiel wird der kleinste und größte gültige Wert (minInclusive bzw. maxInclusive) des neuen Typen [short](#) gegenüber dem Basistypen beschränkt.

Beispiel 27: Einschränkende Spezialisierung eines simplen Typen



```
(1)<xsd:simpleType name="short" id="short">
(2)      <xsd:restriction base="xsd:int">
(3)          <xsd:minInclusive value="-32768"
(4)              id="short.minInclusive"/
(5)      >
(6)          <xsd:maxInclusive value="32767"
(7)              id="short.maxInclusive"/>
(8)      </xsd:restriction>
(9)</xsd:simpleType>
```

Die Bildung aggregierter Typen folgt demselben Muster. Jedoch tritt an die Stelle der Ableitung die Spezifikation des Aggregationstyps (im Beispiel *Liste*) und Angabe des Inhaltstyps (im Beispiel [string](#)).

Beispiel 28: Bildung eines Aggregationstypen



```
(1)<xsd:simpleType name="WarenkorbElemente">
(2)      <xsd:list itemType="xsd:string"/>
(3)</xsd:simpleType>
```

Nachfolgend sind die verschiedenen Beschränkungsmöglichkeiten zusammengefaßt:

- [length](#)

Längenbeschränkung bei [atomaren Typen](#) bzw. Beschränkung der Elementanzahl bei aggregierten Typen.

Längenbeschränkung eines simplen Typs:

```
(1)<xs:simpleType name="Postleitzahl">
(2)      <xs:restriction base="xs:string">
(3)          <xs:length value="5"/>
(4)      </xs:restriction>
(5)</xs:simpleType>
```

Beschränkung der Elementanzahl einer Liste:

```
(1)<xs:simpleType name="VornamenList">
(2)      <xs:list itemType="xs:token"/>
(3)</xs:simpleType>
(4)<xs:simpleType name="VornamenRestrictedList">
(5)      <xs:restriction base="VornamenList">
(6)          <xs:length value="5"/>
(7)      </xs:restriction>
(8)</xs:simpleType>
```

- [minLength](#)

Minimale Länge (bei [atomaren Typen](#) bzw. minimale Elementanzahl bei [aggregierten Typen](#)).

Beispiel (der aggregierte Typ `VornamenRestrictedList` muß mindestens einen Eintrag enthalten):

```
(1)<xs:simpleType name="VornamenList">
(2)      <xs:list itemType="xs:token"/>
(3)</xs:simpleType>
(4)<xs:simpleType name="VornamenRestrictedList">
```

```

(5)    <xs:restriction base="VornamenList">
(6)        <xs:minLength value="1"/>
(7)    </xs:restriction>
(8)</xs:simpleType>

```

Beispiel (der spezialisierte atomare Typ Titel muß aus mindestens fünf Zeichen bestehen):

```

(1)<xs:simpleType name="Titel">
(2)    <xs:restriction base="xs:string">
(3)        <xs:minLength value="5"/>
(4)    </xs:restriction>
(5)</xs:simpleType>

```

- [maxLength](#)

Maximale Länge bei [atomaren Typen](#) bzw. maximale Elementzahl bei [aggregierten Typen](#).

Beispiel (der aggregierte Type VornamenRestrictedList muß mindestens einen, jedoch höchstens drei, Einträge enthalten):

```

(1)<xs:simpleType name="VornamenList">
(2)    <xs:list itemType="xs:token"/>
(3)</xs:simpleType>
(4)<xs:simpleType name="VornamenRestrictedList">
(5)    <xs:restriction base="VornamenList">
(6)        <xs:minLength value="1"/>
(7)        <xs:maxLength value="3"/>
(8)    </xs:restriction>
(9)</xs:simpleType>

```

Beispiel (der spezialisierte atomare Typ Titel muß aus mindestens fünf, darf jedoch aus höchstens 80 Zeichen bestehen):

```

(1)<xs:simpleType name="Titel">
(2)    <xs:restriction base="xs:string">
(3)        <xs:minLength value="5"/>
(4)        <xs:maxLength value="80"/>
(5)    </xs:restriction>
(6)</xs:simpleType>

```

- [pattern](#)

Erlaubt die Inhaltsdefinition durch Muster (reguläre Ausdrücke).

XML-Strukturen sind nicht durch reguläre Ausdrücke darstellbar, hierfür können lediglich die vorgestellten Strukturierungsprimitive eingesetzt werden; daher sind Muster auf atomare Typen beschränkt.

Die [XML-Schemaspezifikation definiert](#) einzelne Symbole und Symbolsequenzen, sowie einige abkürzende Schreibweisen zum Aufbau beliebiger Ausdrücke. Alle anderen Zeichen können direkt in die Musterspezifikation übernommen werden.

Die aus der Verarbeitung regulärer Ausdrücke mit anderen Programmiersprachen oder Werkzeugen bekannten Quantifier stehen in der bekannten Semantik zur Verfügung.

Sei S eine Zeichenkette aus einer beliebigen Anzahl von Zeichen (d.h. sie kann auch leer sein!), dann gilt:

Tabelle 8: Übersicht der Quantifier

Quantifiziertes Mustersymbol	Bedeutung
S	Alle Zeichenketten, die S genau entsprechen.
S?	Alle Zeichenketten, die S genau entsprechen oder die leere Zeichenkette.
S*	Alle Reihungen von S; insbesondere entspricht S* auch der leere Zeichenkette.
S+	Alle Reihungen von S, die S mindestens einmal enthalten. Entspricht der Formulierung: S S*



$S\{n, m\}$	Alle Zeichenketten, die aus mindestens n , jedoch höchstens m Auftreten von S bestehen. Durch $n=0$ lassen sich somit optionale, nach oben begrenzte, Auftrittsanzahlen realisieren, sowie durch $n=m=0$ die leere Zeichenkette ausdrücken
$S\{n\}$	Alle Zeichenketten, die aus genau n Auftreten von S bestehen. Entspricht der Formulierung: $S\{n, n\}$
$S\{n, \}$	Alle Zeichenketten, die aus mindestens n Auftreten von S bestehen. Entspricht der Formulierung: $S\{n\} S^*$

Zur Darstellung nicht-druckbarer Zeichen oder von Metasymbolen werden folgende Fluchtsymbole angeboten:

Tabelle 9: Übersicht der Escape-Symbole

Mustersymbol (escape character)	ausgedrücktes Zeichen
<code>\n</code>	Zeilenumbruch (#xA)
<code>\r</code>	Zeilenvorschub (#xD)
<code>\t</code>	Tabulator (#x9)
<code>\\</code>	<code>\</code>
<code>\ </code>	<code> </code>
<code>\.</code>	<code>.</code>
<code>\-</code>	<code>-</code>
<code>\^</code>	<code>^</code>
<code>\?</code>	<code>?</code>
<code>*</code>	<code>*</code>
<code>\+</code>	<code>+</code>
<code>\{</code>	<code>{</code>
<code>\}</code>	<code>}</code>
<code>\(</code>	<code>(</code>
<code>\)</code>	<code>)</code>
<code>\[</code>	<code>[</code>
<code>\]</code>	<code>]</code>

Ferner sind noch eine Reihe häufig benötigter Zeichenfamilien definiert und in den Kategorien Buchstaben (Letter), Marken (Marks), Zahlen (Numbers), Interpunktion (Punctuation), Trennsymbole (Separators) sowie sonstige (Others) zusammengefaßt. Die Zeichenfamilien innerhalb dieser Kategorien entsprechen den in Unicode v3.1 definierten [general categories](#) in Namen und Aufbau.

Tabelle 10: Buchstaben

symbolische Darstellung	Bedeutung
<code>L</code>	(All Letters) Alle Buchstaben
<code>Lu</code>	(Uppercase) Alle Großbuchstaben
<code>Li</code>	(Lowercase) Alle Kleinbuchstaben
<code>Lt</code>	(Titlecase) Sprachabhängige (potentielle) Großschreibung des ersten Buchstabens eines Wortes. (vgl. UNICODE technical report #21 sowie Derived General Category Lt).
<code>Lm</code>	(Modifiers) Zusammenfassung der verschiedensten Ton- und Betonungszeichen

Lo	(Others) Zusammenfassung von Zeichen, die in keine der sonstigen L-Zeichenfamilien fallen

Tabelle 11: Markierungssymbole

symbolische Darstellung	Bedeutung
M	(All Marks) Alle Markierungssymbole
Mn	(Non-Spacing) Markierungssymbole, ausschließlich Leerzeichen
Mc	(Space combining) Markierungssymbole mit Leerzeichen kombiniert
Me	(Enclosing) Markierungssymbole, die andere Zeichen umschließen

Tabelle 12: Zahlen

symbolische Darstellung	Bedeutung
N	(All Numbers) Beliebige Ziffer; daher nicht notwendigerweise arabisch. Unicode stellt eingekreiste (#x2460-#x2473), geklammerte (#x2474-#x2487) sowie Ordinalzahlen (mit abschließendem Punkt) (#x2488-#x249b) zur Verfügung.
Nd	(Decimal Digit) eine arabische Ziffer
NI	(Letter) auf Buchstaben beruhende Zifferndarstellungen, wie römische Ziffernsymbole (#x20dd-#x217f)
No	(Other) Alle sonstigen Ziffernsymbole

Tabelle 13: Interpunktionszeichen

symbolische Darstellung	Bedeutung
P	(Punctuation) Alle Interpunktionsymbole
Pc	(Connector) Verbindende Interpunktionsymbole (z.B. Unterstrich (#x5f))
Pd	(Dash) Verschiedene Verbindungsstriche
Ps	(Open) Öffnende Bereichssymbole wie die verschiedenen Klammertypen
Pe	(Close) Schließende Bereichssymbole wie die verschiedenen Klammertypen
Pi	(Initial Quote) Öffnende Anführungszeichen. In einigen Fällen Verhalten identisch zu Ps oder Pe
Pf	(Final Quote) Schließende Anführungszeichen. In einigen Fällen Verhalten identisch zu Ps oder Pe
Po	(Other) Alle anderen Interpunktionsymbole

Tabelle 14: Separatoren

symbolische Darstellung	Bedeutung
Z	Alle Separatoren
Zs	(Space) Trennende Leerzeichen
Zl	(Line) Zeilentrenner (#x2028)
Zp	(Paragraph) Absatztrenner (#x2029)

Tabelle 15: Symbole

symbolische Darstellung	Bedeutung
S	Alle Symbole
Sm	(Math) Verschiedene mathematische Symbole (Operatoren, Pfeile, etc.)
Sc	(Currency) Währungssymbole (Dollarzeichen: #x24, Eurozeichen: #x20ac)
Sk	(Modifier) Ton- und Betonungssymbole (ähnlich zu Lm)
So	(Other) Alle anderen Symbole

Tabelle 16: Sonstige Zeichen

symbolische Darstellung	Bedeutung
C	Alle sonstigen
Cc	(Control) Nicht-druckbare Kontrollzeichen
Cf	(Format) Formatierungszeichen (z.B. syrisches Abkürzungssymbol)
Co	(Private Use) ungefähr 137500 Zeichen zur freien anwenderdefinierten Belegung
Cn	(Not Assigned) Zeichen, denen innerhalb Unicode explizit keine Belegung zugewiesen wurde

Reguläre Ausdrücke können innerhalb des pattern-Elements direkt angegeben werden. Die Zeichenkettenfamilien werden durch ihre symbolische Darstellung, eingeleitet durch \p und durch geschweifte Klammern umschlossen, dargestellt. Zusätzlich ist durch \P das Komplement zu jeder der aufgeführten Zeichenkettenfamilien definiert. Ergänzend kann die Definition der zulässigen Zeichengruppen vollständig wahlfrei erfolgen. Hierzu wird das Negationssymbol ^ angeboten, welches eine Aufzählung von Zeichen von der Verwendung ausschließt. Darüberhinaus können auf der Basis simpler Mengendifferenzoperationen eigene Zeichenklassen komfortabel definiert werden. Für die am häufigsten benötigten Zeichenklassen sind durch XML-Schema bereits vorgegebene abkürzende Schreibweisen definiert. Hierzu werden bereits die bisher vorgestellten Syntaxmechanismen angewendet.

Tabelle 17: Zeichensequenzen

abkürzende Schreibweise	entsprechende Langform
.	[^\n\r]
\s	[#\t\n\r]
\S	[^s]
\i	Die initialen Zeichen eines gültigen XML-Namens; entspricht dem ersten Teil der Syntaxproduktion 5
\I	[^i]
\c	Diejenigen Zeichen, die der XML-Syntaxproduktion 4 entsprechen
\C	[^c]
\d	\p{Nd}

\D	[^\d]
\w	[#x0000-#x10FFFF] - [\p{P}\p{S}\p{C}] Alle Zeichen, außer der Klassen für Interpunktions- und Separatorenzeichen, sowie der Klasse der sonstigen Zeichen.
\W	[^\w]

Beispiel (eine vereinfachte, d.h. unter Nicht-Berücksichtigung von Behörden- und Diplomatennummern) deutsche Autonummer im bekannten Format: Einführende Bezeichnung des Landkreises durch mindestens einen, jedoch höchstens drei Großbuchstaben, gefolgt von einem trennenden Bindestrich, an den sich ein oder zwei weitere Großbuchstaben anschließen. Auf ein Leerzeichen folgen eine, jedoch höchstens vier Ziffern:

```
(1)<xs:simpleType name="gerAutoNumber">
(2)  <xs:restriction base="xs:string">
(3)    <xs:pattern value="\p{Lu}{1,3}-\p{Lu}
{1,2} \p{Nd}{1,4}"/>
(4)  </xs:restriction>
(5)</xs:simpleType>
```

Weitere Informationen: [Anhang F -- Reguläre Ausdrücke -- der XML-Spezifikation](#)

- [enumeration](#)

Festlegung von zulässigen Werten eines Typs durch vollständige Aufzählung.

Beispiel (die Ampelfarben: rot, gelb, grün):

```
(1)<xs:simpleType name="ampelfarben">
(2)  <xs:restriction base="xs:string">
(3)    <xs:enumeration value="rot"/>
(4)    <xs:enumeration value="gelb"/>
(5)    <xs:enumeration value="grün"/>
(6)  </xs:restriction>
(7)</xs:simpleType>
```

- [whitespace](#)

Behandlung von white spaces innerhalb eines Elements des definierten Typs. Erlaubte Belegungen und ihre Bedeutung:

preserve: Keine Veränderung des Inhaltes durch Normalisierung; evtl. enthaltene white spaces bleiben erhalten

replace: Alle Vorkommen von Tabulatoren, Zeilenvorschub und Wagenrücklauf werden durch Leerzeichen (#x20) ersetzt

collapse: Nach der Substitution gemäß *replace* werden mehrfach auftretende Leerzeichen zu einem einzigen kompaktifiziert, sowie führende und abschließende Leerzeichen entfernt.

(in [Spezifikation nachschlagen](#))

- [maxInclusive](#)

Höchster zulässiger numerischer Wert. Im mathematischen Sinne sind daher alle Werte kleiner oder gleich dem im value-Attribut angegebenen zugelassen.

Beispiel (Zahlen <= 100):

```
(1)<xs:simpleType name="uHu">
(2)  <xs:restriction base="xs:decimal">
(3)    <xs:maxInclusive value="100"/>
(4)  </xs:restriction>
(5)</xs:simpleType>
```

Anmerkung: In vielen Fällen kann eine maxInclusive-Festlegung ohne Informationsverlust in eine äquivalente maxExclusive-Definition überführt werden.

- [maxExclusive](#)

Wert „unterhalb“ (im mathematischen Sinne „kleiner“) dem alle numerischen Belegungen zugelassen sind.

Beispiel (Zahlen < 100):

```
(1)<xs:simpleType name="uHu">
(2)  <xs:restriction base="xs:decimal">
(3)    <xs:maxExclusive value="100"/>
(4)  </xs:restriction>
```

(5)</xs:simpleType>

Als Belegungen des Typs uHu sind alle Zahlen kleiner als 100 zugelassen. Durch die Verwendung des XSD-Typen [decimal](#) als Basistyp kann auch keine verlustfreie Überführung in eine maxInclusive-Festlegung überführt werden, da hierfür die größte zugelassene Zahl fixiert werden müsste, was jedoch die Typdefinition von [decimal](#) explizit offen lässt.

- [minInclusive](#)

Kleinster zugelassener Wert.

Beispiel (Zahlen >= 25):

```
(1)<xs:simpleType name="uFz">
(2)   <xs:restriction base="xs:decimal">
(3)       <xs:minInclusive value="25"/>
(4)   </xs:restriction>
(5)</xs:simpleType>
```

- [minExclusive](#)

Kleinster Wert, der nicht mehr zugelassen ist. Im mathematischen Sinne sind alle Zahlen, die größer sind, gültige Belegungen.

Beispiel (Zahlen > 25):

```
(1)<xs:simpleType name="uFz">
(2)   <xs:restriction base="xs:decimal">
(3)       <xs:minExclusive value="25"/>
(4)   </xs:restriction>
(5)</xs:simpleType>
```

- [totalDigits](#)

Gesamtstellen einer Zahl, gebildet aus der Summe der Vorkomma- und der Nachkommastellen.

Beispiel (Zahlen mit höchstens sieben Stellen):

```
(1)<xs:simpleType name="myNumber">
(2)   <xs:restriction base="xs:decimal">
(3)       <xs:totalDigits value="7"/>
(4)   </xs:restriction>
(5)</xs:simpleType>
```

- [fractionDigits](#)

Anzahl der Nachkommastellen eines Dezimalbruches.

Beispiel (Zahl mit höchstens neuen Stellen, davon zwei Nachkommastellen):

```
(1)<xs:simpleType name="myNumber">
(2)   <xs:restriction base="decimal">
(3)       <xs:totalDigits value="9"/>
(4)       <xs:fractionDigits value="2"/>
(5)   </xs:restriction>
(6)</xs:simpleType>
```

Definition von Attributen:

Die Attributdeklaration erfolgt durch das XSD-Element [attribute](#). Die Mächtigkeit entspricht auch hier, wie bereits für die Elemente verwirklicht, einer Obermenge der DTD. So können neben optionalen, zwingenden und konstanten Attributen auch Aufzählungsattribute und Mengen realisiert werden. Hierbei wurde auf die Orthogonalität zum durch simpleType geschaffenen Typmechanismus geachtet.

Die Charakteristika (ausgedrückt in Attributen des XSD-Elements attribute) einer Attributdeklaration umfassen:

- **name:** Ein Doppelpunkt-freier Namen (NCName) gemäß [Namensraumproduktion 7](#).
- **id:** erlaubt die eindeutige Kennzeichnung eines Attributs durch eine Schema-weit eindeutige Zeichenkette.
- **default:** Belegung mit Vorgabewert.
- **fixed:** Konstante Belegung.
- **type:** Typ des Attributs, definiert durch einen simpleType.
- **form:** Legt fest, ob der Attributname im XML-Instanzdokument durch ein Namensraumpräfix eingeleitet wird (Belegung: qualified, andernfalls unqualified).
- **ref:** Verweis auf eine globale Attributdefinition.

- **use:** Verwendung des Attributes, Wert entspricht optional, required oder prohibited. Vorgabegemäß wird optional angenommen und das Attribut damit nicht zwingend im XML-Dokument erwartet. Den Gegensatz hierzu bildet required, wodurch das Attribut als zwingend anzugeben definiert wird. prohibited verbietet die Nutzung des Attributes im XML-Dokument.

Anmerkung: Einen Anwendungsfall der Belegung prohibited für use bilden Attribute, die innerhalb des Schemas bereits definiert sind, jedoch noch nicht zur allgemeinen Nutzung freigegeben wurden.

Beispiel 29: Einige Attributdefinitionen

```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
(3)    <xsd:attribute name="myAtt1"/>
(4)
(5)    <xsd:attribute name="myAtt2" type="xsd:decimal"/>
(6)
(7)    <xsd:attribute name="myAtt3">
(8)        <xsd:simpleType>
(9)            <xsd:restriction base="xsd:int">
(10)                <xsd:minInclusive value="10"/>
(11)                <xsd:maxInclusive value="20"/>
(12)            </xsd:restriction>
(13)        </xsd:simpleType>
(14)    </xsd:attribute>
(15)
(16)    <xsd:simpleType name="myType1">
(17)        <xsd:restriction base="xsd:string">
(18)            <xsd:maxLength value="5"/>
(19)        </xsd:restriction>
(20)    </xsd:simpleType>
(21)    <xsd:attribute name="myAtt4" type="myType1"/>
(22)
(23)
(24)
(25)    <xsd:element name="foo">
(26)        <xsd:complexType>
(27)            <xsd:attribute ref="myAtt1" use="optional"/>
(28)            <xsd:attribute ref="myAtt2" use="required"/>
(29)            <xsd:attribute ref="myAtt3" use="prohibited"/>
(30)            <xsd:attribute ref="myAtt4"/>
(31)            <xsd:attribute name="myAtt5" type="xsd:date" id="myDate"/>
(32)            <xsd:attribute name="myAtt6">
(33)                <xsd:simpleType>
(34)                    <xsd:restriction base="xsd:float">
(35)                        <xsd:totalDigits value="5"/>
(36)                    </xsd:restriction>
(37)                </xsd:simpleType>
(38)            </xsd:attribute>
(39)        </xsd:complexType>
(40)    </xsd:element>
(41)
(42)</xsd:schema>
```



Download des Beispiels

Das Beispiel zeigt einige Varianten der Attributdeklaration. So definieren myAtt1 mit myAtt4 globale Attribute, die innerhalb verschiedener Elemente verwendet werden können. Hierdurch wird die bereits für Elemente verwirklichte Mimik der einmaligen Deklaration und anschließenden beliebigen Verwendung auch auf Attribute ausgedehnt. Die Nutzung der so deklarierten Attribute geschieht durch das ref-Attribut innerhalb des Attribute-Elements des beherbergenden Elements. myAtt1 definiert ein typenloses Attribut, dem vorgabegemäß der allgemeinste Typ [anyType](#) zugeordnet wird. Die Angabe dieses Attributes ist optional (use="optional"), was der Vorgabe entspricht.

Der XSD-Standardtyp [decimal](#) findet zur Definition des Attributs myAtt2 Verwendung. Die zwingend anzugebenden (use="required") Inhalte dieses Attributs werden durch einen XML-Schema-Parser auf Typkonformität geprüft.

myAtt3 veranschaulicht die Bildung eines anonymen (inneren) atomaren Typen zur Definition eines Attributs. Der durch Restriktion gebildete neue Datentyp steht ausschließlich innerhalb des Attributs myAtt3 zur Verfügung. Die Syntax der Datentypspezialisierung entspricht der im vorhergehenden Abschnitt diskutierten. Zudem ist die Verwendung des Attributes innerhalb eines

XML-Dokumentes untersagt; ausgedrückt durch die Belegung `use="prohibited"`

Analog der Typisierung eines Elementinhaltes durch einen anwenderdefinierten Typen gestaltet sich das Vorgehen für Attribute. Veranschaulicht wird dies durch die Definition von `myAtt4`. Sie greift auf den eigen-definierten Typen `myType1` zurück.

Dem Attribut `myAtt5` ist zusätzlich zur Benennung, die innerhalb des verwendenden Elementes eindeutig sein sollte, ein Dokument-weiter Schlüssel (`id`) zugeordnet.

Innerhalb des Elements `foo` werden die fünf zuvor definierten Attribute verwendet. Trotz der Reihenfolge der Definitionen im `complexType`-Element verfügen die Attribute im XML-Instanzdokument -- auch bei der Verwendung von XML-Schema -- über keinerlei Reihenfolge ([vgl. XML-Spezifikation](#)).

Zusätzlich enthält die Elementdefinition für `foo` mit `myAtt6` ein „lokales“ Attribut. Diese Definitionsvariante entspricht am ehesten der der Document Type Definition, da sie eine Wiederverwendung außerhalb des definierenden Elements ausschließt.

Beispiel 30: Vollständiges XML-Schema der Projektverwaltung

```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
(3)  <xsd:element name="Nachname" type="xsd:string"/>
(4)  <xsd:complexType name="PersonType">
(5)    <xsd:sequence>
(6)      <xsd:element ref="Vorname" maxOccurs="unbounded"/>
(7)      <xsd:element ref="Nachname" maxOccurs="unbounded"/>
(8)      <xsd:element name="Qualifikationsprofil"
type="QualifikationsprofilType" minOccurs="0"/>
(9)    </xsd:sequence>
(10)   <xsd:attribute name="PersID" type="xsd:ID" use="required"/>
(11)   <xsd:attribute name="Gehaltsgruppe" default="1a">
(12)     <xsd:simpleType>
(13)       <xsd:restriction base="xsd:NMTOKEN">
(14)         <xsd:enumeration value="1"/>
(15)         <xsd:enumeration value="1a"/>
(16)         <xsd:enumeration value="2"/>
(17)       </xsd:restriction>
(18)     </xsd:simpleType>
(19)   </xsd:attribute>
(20)   <xsd:attribute name="mitarbeitInProjekt" type="xsd:IDREFS"
use="required"/>
(21) </xsd:complexType>
(22) <xsd:complexType name="ProjektType">
(23)   <xsd:attribute name="ID" type="xsd:ID" use="required"/>
(24)   <xsd:attribute name="date" type="xsd:date"/>
(25)   <xsd:attribute name="budget" default="10000.00">
(26)     <xsd:simpleType>
(27)       <xsd:restriction base="xsd:double">
(28)         <xsd:fractionDigits value="2"/>
(29)       </xsd:restriction>
(30)     </xsd:simpleType>
(31)   </xsd:attribute>
(32)   <xsd:attribute name="Projektleiter" type="xsd:IDREF" use="required"/>
(33)   <xsd:attribute name="Mitarbeiter" type="xsd:IDREFS" use="required"/>
(34) </xsd:complexType>
(35) <xsd:element name="ProjektVerwaltung">
(36)   <xsd:complexType>
(37)     <xsd:sequence>
(38)       <xsd:element name="Person" type="PersonType"
maxOccurs="unbounded"/>
(39)       <xsd:element name="Projekt" type="ProjektType"
maxOccurs="unbounded"/>
(40)     </xsd:sequence>
(41)     <xsd:attribute name="version" type="xsd:string" fixed="1.0"/>
(42)   </xsd:complexType>
(43) </xsd:element>
(44) <xsd:complexType name="QualifikationsprofilType" mixed="true">
(45)   <xsd:choice minOccurs="0" maxOccurs="unbounded">
(46)     <xsd:element ref="Qualifikation"/>
(47)     <xsd:element ref="Leistungsstufe"/>
(48)     <xsd:any namespace="http://www.w3.org/1999/xhtml"/>
(49)   </xsd:choice>
(50) </xsd:complexType>
(51) <xsd:element name="Qualifikation" type="xsd:string"/>
(52) <xsd:element name="Leistungsstufe" type="xsd:string"/>
(53) <xsd:element name="Vorname" type="xsd:string"/>
```



(54)</xsd:schema>

Download des Beispiels

Abschließend eine gültige (sowohl *valid* als auch *schema valid*) Dokumentinstanz der Projektverwaltungsstruktur.

Beispiel 31: Gültiges Projektverwaltungsdokument

```
(1)<?xml version="1.0" encoding="ISO-8859-1"?>
(2)<ProjektVerwaltung
(3)    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(4)    xsi:noNamespaceSchemaLocation="http://www.jeckle.de/vorlesung/xml/examples/
projektverwaltung.xsd">
(5)    <Person PersID="Pers01" mitarbeitInProjekt="Prj01">
(6)        <Vorname>Hans</Vorname>
(7)        <Nachname>Hinterhuber</Nachname>
(8)    </Person>
(9)    <Person PersID="Pers02" mitarbeitInProjekt="Prj02">
(10)        <Vorname>Franz</Vorname>
(11)        <Vorname>Xaver</Vorname>
(12)        <Nachname>Obermüller</Nachname>
(13)        <Qualifikationsprofil>
(14)            IT-Kompetenz verschiedene Betriebssysteme und
<Leistungsstufe>professionelle</Leistungsstufe>
(15)            <Qualifikation>Programmierung</Qualifikation> verschiedener
Programmiersprachen
(16)        <Qualifikation>Entwickler</Qualifikation> von 1988-1990
(17)        <Qualifikation>Projektleiterfunktion</Qualifikation> von 1990-93 im X42-
Projekt in Abteilung AB&C
(18)    </Qualifikationsprofil>
(19) </Person>
(20) <Person PersID="Pers03" mitarbeitInProjekt="Prj02">
(21)     <Vorname>Fritz</Vorname>
(22)     <Nachname>Meier</Nachname>
(23) </Person>
(24) <Projekt ID="Prj01" Projektleiter="Pers01" Mitarbeiter="Pers01"/>
(25) <Projekt ID="Prj02" Projektleiter="Pers02" Mitarbeiter="Pers03"/>
(26)</ProjektVerwaltung>
```



Download des Beispiels

Werkzeuge:

Zwar existiert -- wie für alle XML-Dokumente -- die Möglichkeit, Dokument Typ Definitionen und XML-Schemata „per Hand“ mit einem Texteditor zu erstellen, jedoch ist dieses Vorgehen, insbesondere für umfangreiche XML-Vokabulare, zeitaufwendig und fehlerträchtig. Zusätzlich läßt die rein textuelle Formulierung die entstehenden Schemadokumente schnell unübersichtlich werden. Inzwischen existieren einige gute DTD- und Schemaeditoren, die zumeist neben visueller Syntaxhervorhebung auch die kontextsensitive Editierung erlauben und so eine wesentliche Erleichterung der Schemaerzeugung bilden. Gleichzeitig bieten die meisten verfügbaren Werkzeuge dieser Klasse auch Möglichkeiten zur Validierung des erzeugten Schemas an. Ergänzend wird vielfach auch eine graphische Repräsentation der DTD- oder XSD-Struktur angeboten.

Die Abbildungen zeigen Ansichten der Werkzeuge [XML Authority](#) bzw. [XML Spy](#)



Abbildung 16:
XML Authority

(click on image to
enlarge!)



Abbildung 17:
XML Spy (click

on image to
enlarge!)



Abbildung 18:
XML Spy (click

on image to
enlarge!)

Web-Referenzen 6: Weiterführende Links und Werkzeuge



- [XML Schema Part 0: Primer](#)
 - [XML Schema Part 1: Structures](#)
 - [XML Schema Part 2: Datatypes](#)
 - [XML Schema @ Cover-Pages](#)
 - [Parsing the Atom](#) -- Diskussion über die Vor- und Nachteile inhärent komplexer atomarer Typen
 - [Schema-Informationen @ jeckle.de](#)
 - [XML-Authority \(DTD- und XSD-Editor\)](#)
 - [XML Spy \(DTD- und XSD-Editor\)](#)
-
-

Die Lokatorsprache XPath

Zur Extraktion beliebiger Teile eines wohl-geformten XML-Dokuments verabschiedete das W3C 1999 die Sprache *XPath*. Sie bildet eine pfadorientierte *Lokatorsprache*, die das Auffinden von Dokumentteilen (einzelnen Elementen, Attributen, etc.) durch Pfadausdrücke, die sich an der Struktur des XML-Dokuments orientieren, gestattet.

Die Grenze zwischen Lokatorsprache und „echter“ Anfragesprache wie SQL sind fließend. Zwei Unterscheidungsmerkmale sollen jedoch hervorgehoben werden: XPath wird im üblichen Anwendungsfall nicht interaktiv oder in eine Programmiersprache als Wirtssprache eingebettet verwendet, sondern wurde (zunächst) nur für die Nutzung in Kombination mit der Transformationssprache *XSLT* und den erweiterten Verweisen der Sprache *XPointer* konzipiert. Zum zweiten fehlt XPath die üblicherweise mit dem reinen Anfrageteil verwobene Manipulationssprache zur Änderung bereits bestehender Daten; XPath ist allein für den lesenden Zugriff auf XML-Dokumente ausgelegt.

Hinweis: XPath unterscheidet XML-üblich zwischen Groß- und Kleinschreibung. Daher sind Element- und Attributnamen unbedingt in der im Dokument gewählten Schreibweise anzugeben.

Lokalisierungspfade:

Lokalisierungspfade dienen der abstrakten Beschreibung einer Menge von Informationsknoten innerhalb eines Dokuments.

Die einfachste Form eines Lokalisierungspfades beschreibt der *Wurzellokalisierungspfad* (*root location path*), ausgedrückt durch „/“. Er liefert für jedes XML-Dokument den Wurzelknoten. Dieser ist nicht identisch mit dem Wurzelelement eines XML-Dokuments! Der (unbenannte) Wurzelknoten entspricht dem [Document Information Item](#) des Information Sets, während das erste benannte Element des Dokuments durch ein [Element Information Item](#) dargestellt wird.

Die Navigation zu den einzelnen Elementknoten, oder Knotenmengen, wird durch einen Pfadausdruck realisiert. Die explizite Variante erlaubt die Angabe aller zu traversierenden Knoten bis hin zu den zu extrahierenden. Hierzu werden die Knoten, von der Wurzel absteigend durch „/“-Symbole separiert, notiert. Wegen der Korrespondenz der voneinander abgetrennten Knotennamen und den Baumstufen, werden diese auch als *Lokalisierungsschritte* bezeichnet. Als weitere sprachliche Analoge spiegelt der XPath-Ausdruck, von links nach rechts gelesen, auch die Schritte -- ausgehend vom Wurzelelement des Dokuments -- zur Lokalisierung der gesuchten Knotenmenge wieder.

Das Beispiel zeigt eine solche Definition am [Beispiel der Projektverwaltung](#).

Anmerkung: Das Resultat ist in XML-Notation dargestellt, obwohl genaugenommen eine Knotenmenge des Information Sets als Resultat zurückgeliefert wird. Die gewählte XML-Darstellung ist hierbei nur eine der möglichen Varianten zur Ergebnispräsentation.

Beispiel 32: XPath-Ausdruck zur Lokalisierung aller Vornamen

XPath-Ausdruck: /ProjektVerwaltung/Person/Vorname



Ergebnis: <Vorname>Hans</Vorname>,
<Vorname>Franz</Vorname>,
<Vorname>Xaver</Vorname>,
<Vorname>Fritz</Vorname>

Die Einzelknoten werden entsprechend ihrer Auftrittsreihenfolge im Quelldokument (sog. *document order*) zurückgegeben.

Die expliziten Pfadausdrücke lassen sich in beliebiger Länge fortsetzen, jedoch zeigen sie fundamentale Schwächen in Puncto Flexibilität. Wie im [Beispiel der XHTML-Verwendung innerhalb eines eigenen XML-Dokuments](#) gesehen, kann Information desselben Typs (d.h. umschlossen durch denselben Tag) verschiedene Elternknoten besitzen. So im Beispiel, dort ist die Qualifikation auf derselben Baumstufe sowohl unterhalb des Elternelements em als auch u anzutreffen.

Als Lösung erlaubt XPath die Nutzung von Platzhaltern statt der expliziten Elementnamen innerhalb

eines Lokalisierungsschrittes. In der Folge entstehen freie Lokalisierungsschritte, die alle Kindknoten einer im direkt vorhergehenden Lokalisierungsschritt selektierten Knotenmenge adressieren.

Der nachfolgende XPath-Ausdruck zeigt dies am Beispiel des Qualifikationsprofils.

Beispiel 33: Platzhalter in Lokalisierungsschritten



XPath-Ausdruck: /ProjektVerwaltung/Person/Qualifikationsprofil/*/Qualifikation

Ergebnis: <Qualifikation>Programmierung</Qualifikation>
<Qualifikation>Projektleiterfunktion</Qualifikation>

Der Pfadausdruck liefert die beiden Kindelemente Qualifikation -- unabhängig von der Benennung des Elternknotens -- die direkt unterhalb des Knotens Qualifikationsprofil angeordnet sind.

Allerdings enthält die Ausgabe nicht alle Knoten des Typs Qualifikation. Der gegebene Pfadausdruck gestattet lediglich das Überspringen einer Hierarchieebene. Daher wird der hierarchisch tieferstehende Qualifikations-Knoten mit Inhalt Entwickler nicht lokalisiert. Die (zunächst naheliegende) Lösung den Pfadausdruck zu /ProjektVerwaltung/Person/Qualifikationsprofil/**/Qualifikation zu erweitern liefert nicht das gewünschte Resultat aller Qualifikations-Knoten, sondern ausschließlich den zuvor nicht lokalisierbaren, da der modifizierte Ausdruck nun zwingend zwei freie Lokalisierungsschritte vorsieht.

Zur Variierung der Tiefe der freien Schritte sieht XPath die Schreibweise „//“ vor. Sie erlaubt die Lokalisierung der Kindknoten auf einer beliebigen Hierarchiestufe.

Definition 11: Lokalisierungsschritt

Ein Lokalisierungsschritt setzt sich aus dem Namen der Achse gefolgt von zwei Doppelpunkten und einem *Knotentest*, optional ergänzt um ein auszuwertendes Prädikat, zusammen.



Wird keine Achse spezifiziert, so gilt vorgabegemäß die Achse child.

Ein *Knotentest* ist syntaktisch ein *QName*, der genau dann erfüllt ist, wenn der Knotenname mit dem Namen des Knotentests übereinstimmt.

Das *Prädikat* filtert die Ergebnismenge hinsichtlich verschiedener Charakteristika wie Existenz von Kindknoten oder Attributen, Position in der Ergebnismenge, etc.

Das Beispiel zeigt die korrekte XPath-Formulierung zur Lokation aller Qualifikations-Knoten:

Beispiel 34: Hierarchieunabhängige Knoten-Lokalisierung

XPath-Ausdruck: /ProjektVerwaltung/Person/Qualifikationsprofil//Qualifikation

Ergebnis: <Qualifikation>Programmierung</Qualifikation>



<Qualifikation>Entwickler</Qualifikation>

<Qualifikation>Projektleiterfunktion</Qualifikation>

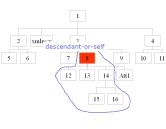
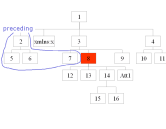
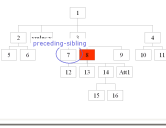
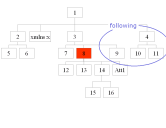
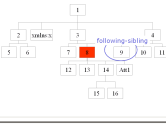
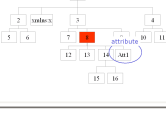
Durch die abkürzende Schreibweise „//“ entsteht ein Muster zur Selektion aller nachfolgenden Knoten. In Verallgemeinerung dieses Konzepts bietet XPath sog. *Achsen* an, um relativ zum aktuellen Knoten beliebige Teilbäume zu lokalisieren.

Die Abbildung zeigt die verschiedenen durch Achsen zugänglichen Knotenmengen relativ zum rot hervorgehobenen aktuellen Knoten.

[Download der XML-Datei](#) mit dem Beispiel der Graphik

Tabelle 18: XPath-Achsen und ihre Bedeutung

Achse	Semantik	Im Beispiel selektierte Knoten	Graphik
self	Lokalisiert den aktuellen Knoten Als abkürzende Schreibweise kann der Punkt „.“ verwendet werden.	XPath-Ausdruck: /node1/node3/node8/self::node8 Ergebnisknotenmenge: {8}	
child	Lokalisiert die (direkten) Kindknoten des aktuellen Knotens	XPath-Ausdruck: /node1/node3/node8/child::* Ergebnisknotenmenge: {12, 13, 14}	

descendant	Lokalisiert transitiv alle Kindknoten des aktuellen Knotens, außer Attribut- und Namensraumknoten	XPath-Ausdruck: /node1/node3/node8/ descendant::* Ergebnisknotenmenge: {12, 13, 14, 15, 16}	
descendant-or-self	Lokalisiert transitiv alle Kindknoten des aktuellen Knotens (außer Attribut- und Namensraumknoten), sowie den Knoten selbst	XPath-Ausdruck: /node1/node3/node8/ descendant-or-self::* Ergebnisknotenmenge: {8, 12, 13, 14, 15, 16}	
parent	Lokalisiert den Elternknoten des aktuellen Knotens, falls existent	XPath-Ausdruck: /node1/node3/node8/ parent::* Ergebnisknotenmenge: {3}	
ancestor	Lokalisiert transitiv alle Elternknoten des aktuellen Knotens. Die ancestor-Achse enthält daher immer den Wurzelknoten, außer der aktuelle Knoten ist es selbst; in diesem Falle liefert die Achse die leere Menge	XPath-Ausdruck: /node1/node3/node8/ ancestor::* Ergebnisknotenmenge: {1, 3}	
ancestor-or-self	Lokalisiert transitiv alle Elternknoten des aktuellen Knotens, sowie den aktuellen Knoten. Diese Achse enthält immer den Wurzelknoten des Dokuments.	XPath-Ausdruck: /node1/node3/node8/ ancestor-or-self::* Ergebnisknotenmenge: {1, 3, 8}	
preceding	Lokalisiert alle dem aktuellen Knoten vorausgehenden Knoten, ohne seine Vorfahren sowie Attribut- und Namensraumknoten	XPath-Ausdruck: /node1/node3/node8/ preceding::* Ergebnisknotenmenge: {2, 5, 6, 7}	
preceding-sibling	Lokalisiert die im Dokument vor dem aktuellen Knoten auftretenden Geschwisterknoten	XPath-Ausdruck: /node1/node3/node8/ preceding-sibling::* Ergebnisknotenmenge: {7}	
following	Lokalisiert alle dem aktuellen Knoten nachfolgenden Knoten ohne dessen Kind-, Attribut und Namensraumknoten	XPath-Ausdruck: /node1/node3/node8/ following::* Ergebnisknotenmenge: {9, 4, 10, 11}	
following-sibling	Lokalisiert alle „Geschwister“ des aktuellen Knotens, d.h. Knoten auf derselben Hierarchieebene.	XPath-Ausdruck: /node1/node3/node8/ following-sibling::* Ergebnisknotenmenge: {9}	
attribute	Lokalisiert Attribut(e) eines Knotens	XPath-Ausdruck: /node1/node3/node8/ attribute::* Ergebnisknotenmenge: {Att1}	
namespace	Lokalisiert Namensraum-Attribut eines Knotens	XPath-Ausdruck: /node1/node3/node8/ namespace::* Ergebnisknotenmenge: {xmlns:xml="http://www.w3.org/XML/1998/namespace", xmlns:x="namespace:www.jeckle.de/vorlesung/xml"}	

Anmerkung:

Die Achsen ancestor, descendant, following, preceding und self partitionieren ein Dokument (unter Auslassung der Attribut- und Namensraumknoten): sie überschneiden sich nicht und enthalten alle Elementknoten des Dokuments.

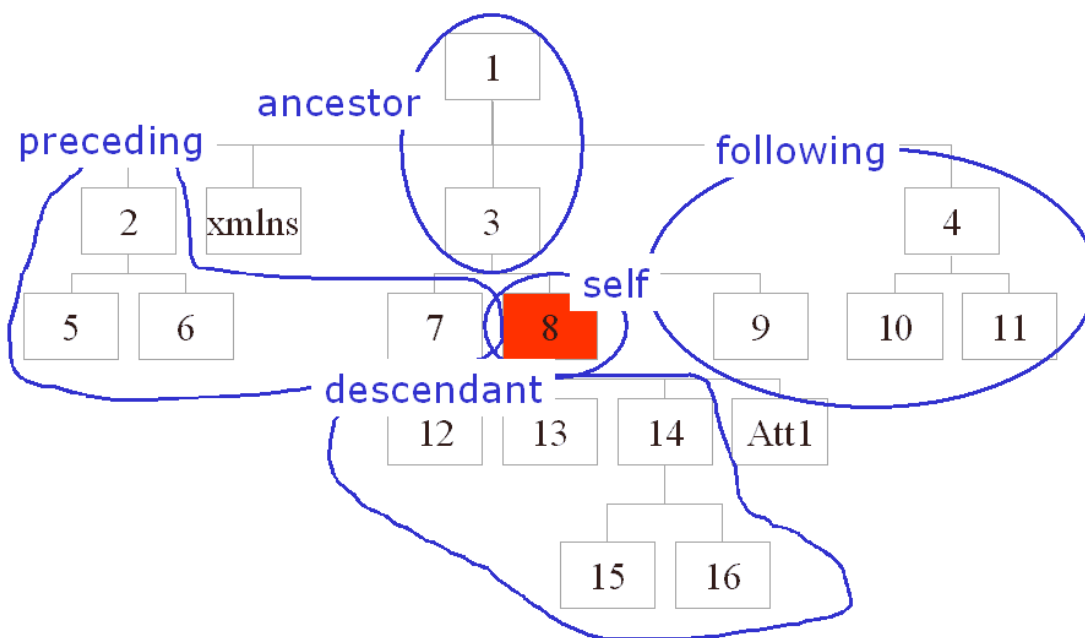


Abbildung 19: Partitionierung eines XML-Dokuments durch XPath-Achsen (click on image to enlarge!)

Filterung durch Prädikate:

Ein -- durch eckige Klammern abgegrenztes -- Prädikat kann innerhalb jedes Lokalisierungsschrittes eines XPath-Ausdrucks angegeben werden. Fehlt es, wird die bisher ermittelte Knotenmenge nicht modifiziert.

Das Prädikat kann selbst ein gültiger XPath-Ausdruck sein.

Das prinzipielle Vorgehen kann folgendermaßen beschrieben werden:

Beginnend von links nach rechts für jeden Lokalisierungsschritt: (1) Ermittlung der zur Anfrage passenden Knotenmenge

(2) Reduzierung der Ergebnismenge um diejenigen Knoten, für die das Prädikat false liefert.

Befinden sich rechts vom aktuell bearbeiteten Lokalisierungsschritt weitere Ausdrücke, so wird die Resultatmenge als Eingabe eines weiteren Schritts (1) übergeben.

Beispiel 35: Selektion unter Anwendung eines Prädikats



XPath-Ausdruck: `//Person[Qualifikationsprofil]/Nachname`

Ergebnis:

`<Nachname>Obermüller</Nachname>`

Der Ausdruck selektiert an beliebiger Stelle des Dokuments („//“) alle Knoten des Typs Person. Die Knotenmenge wird um diejenigen Personen vermindert, zu denen kein Qualifikationsprofil angelegt ist. D.h. Es werden nur diejenigen Knoten selektiert, die über einen Kindknoten des Typs Qualifikationsprofil verfügen. Von dieser Knotenmenge (des Typs Person!) werden anschließend im zweiten Lokalisierungsschritt die Kindknoten des Typs Nachname selektiert. Mithin liefert der XPath-Ausdruck alle Nachnamen von Personen, zu denen ein Qualifikationsprofil abgelegt ist.

Anmerkung: Das Beispiel nutzt im Prädikat die abkürzende Schreibweise zur Angabe der Vorgabeachse child. Die ausführliche Schreibweise -- mit unveränderter Semantik -- des XPath-Ausdrucks lautet daher: `//Person[child::Qualifikationsprofil]/Nachname`

Durch die zusätzliche Definition eines Prädikats für den zweiten Lokalisierungsschritt kann eine weitere Filterung der Ergebnismenge realisiert werden. Zusätzlich können innerhalb eines Prädikats neben XPath-Ausdrücken auch einige vordefinierte Funktionen verwendet werden.

Das Beispiel zeigt die Selektion der Vornamen als Kind eines Personen-Knotens (Test der Elternschaft durch erstes Prädikat), wenn dieser mit „O“ beginnt (Test durch [starts-with](#)-Funktion innerhalb des zweiten Prädikats). Die Struktur der [Eingabedatei](#) zwingt zusätzlich zur Anwendung der [following](#)-Achse, da Knoten des Typs Nachname in der Dokumentreihenfolge nach Knoten des Typs Vornamen auftreten.

Beispiel 36: Schrittweise Berechnung einer Selektion unter Verwendung mehrerer Prädikate

XPath-Ausdruck: `//Person[parent::ProjektVerwaltung]/Vorname[starts-with(following::Nachname, 'O')]`

Ausgewerteter XPath: `//Person`

Ergebnis:

```
<Person PersID="Pers01" mitarbeitInProjekt="Prj01"> ... </
Person>
<Person PersID="Pers02" mitarbeitInProjekt="Prj02"> ... </
Person>
<Person PersID="Pers03" mitarbeitInProjekt="Prj02"> ... </
Person>
```



Ausgewerteter XPath: `//Person[parent::ProjektVerwaltung]`

Ergebnis:

```
<Person PersID="Pers01" mitarbeitInProjekt="Prj01"> ... </
Person>
<Person PersID="Pers02" mitarbeitInProjekt="Prj02"> ... </
Person>
<Person PersID="Pers03" mitarbeitInProjekt="Prj02"> ... </
Person>
```

Ausgewerteter XPath: `//Person[parent::ProjektVerwaltung]/Vorname`

Ergebnis:

```
<Vorname>Hans</Vorname>
<Vorname>Franz</Vorname>
<Vorname>Xaver</Vorname>
<Vorname>Fritz</Vorname>
```

Ausgewerteter XPath: `//Person[parent::ProjektVerwaltung]/Vorname[following::Nachname]`

Ergebnis:

```
<Vorname>Hans</Vorname>
<Vorname>Franz</Vorname>
<Vorname>Xaver</Vorname>
<Vorname>Fritz</Vorname>
```

Ausgewerteter XPath:

`//Person[parent::ProjektVerwaltung]/Vorname[starts-with(following::Nachname, 'O')]`

Ergebnis:

```
<Vorname>Franz</Vorname>
<Vorname>Xaver</Vorname>
```

Die durch die [XPath-Spezifikation](#) vordefinierten Funktionen lauten in der Übersicht:

Tabelle 19: XPath-Funktionen für Knotenmengen (node-sets)

Funktionsprototyp	Funktionalität
<i>number</i> last()	Liefert die Größe der aktuellen Knotenmenge; damit den Index des letzten Elements
<i>number</i> position()	Liefert die Position des aktuellen Knotens innerhalb der Knotenmenge. Die erste Knoten trägt die Positionsnummer 1.
<i>number</i> count(<i>node-set</i>)	Liefert Elementzahl der übergebenen Knotenmenge
<i>node-set</i> id(<i>object</i>)	Liefert denjenigen Knoten, dessen ID-typisiertes Attribut den Argumentwert aufweist. <i>Anmerkung:</i> Zur Nutzung dieser Funktion muß zwingend eine Dokument-Grammatik (DTD oder Schema) zum Eingangsdokument vorliegen.
<i>string</i> local-name (<i>node-set</i> ?)	Liefert den local name (oder die Menge der Namen) der übergebenen Knotenmenge. Wird keine Knotenmenge übergeben, dann wird der aktuelle Knoten als Argument genutzt.



<code>string namespace-uri (node-set?)</code>	Liefert die Namensraum-URI der übergebenen Knotenmenge. Wird keine Knotenmenge übergeben, dann wird der aktuelle Knoten als Argument genutzt. <i>Anmerkung:</i> Handelt es sich nicht um einen Element- oder Attributknoten, so ist die retournierte Zeichenkette leer.
<code>string name (node-set?)</code>	Liefert die QName(n) (=qualifizierte(r) Name(n)) aus Namensraumkürzel und local name der übergebenen Knotenmenge, oder des aktuellen Knotens bei leerer Knotenmenge. <i>Anmerkung:</i> Nur für Element- und Attributknoten liefert name andere Resultate als local-name.

Tabelle 20: XPath-Funktionen für Zeichenketten

Funktionsprototyp	Funktionalität
<code>string string (object)?</code>	Liefert Zeichenkettenrepräsentation einer Knotenmenge. Dabei wird der Zeichenkettenwert des ersten Knotens in der Dokumentreihenfolge zurückgegeben, andernfalls die leere Zeichenkette.
<code>string concat (string, string, string*)</code>	Verkettet mindestens zwei Zeichenketten.
<code>boolean starts-with (string1, string2)</code>	Liefert true falls string1 das zweite Argument string2 als Präfix enthält; andernfalls false
<code>boolean contains (string1, string2)</code>	Liefert true falls string1 die Zeichenkette aus string2 enthält; andernfalls false.
<code>string substring-before (string1, string2)</code>	Liefert denjenigen Teil der Zeichenkette <i>string1</i> , der sich vor dem ersten Auftreten der Zeichenkette <i>string2</i> befindet.
<code>string substring-after (string1, string2)</code>	Liefert denjenigen Teil der Zeichenkette <i>string1</i> , der sich nach dem ersten Auftreten der Zeichenkette <i>string2</i> befindet.
<code>string substring (string, number1, number2?)</code>	Liefert eine Zeichenkette der Länge <i>number2</i> aus <i>string</i> , beginnend mit der Position <i>number1</i> . Fehlt das dritte Argument, so wird der Teilstring bis zum Ende der Zeichenkette <i>string</i> zurückgegeben. <i>Anmerkung:</i> Das erste Zeichen trägt die Indexnummer 1, nicht 0 wie in Java und C üblich.
<code>number string-length(string?)</code>	Liefert die Länge der übergebenen Zeichenkette. Wird kein Argument übergeben, so wird die Länge des zuvor in eine Zeichenkette konvertierten aktuellen Knotens zurückgegeben.
<code>string normalize-space (string?)</code>	Liefert die übergebene Zeichenkette unter Entfernung führender, schließender und mehrfacher Leerzeichen zurück. Ferner werden noch evtl. in der Argumentzeichenkette enthaltenen Entitätsreferenzen aufgelöst. <i>Anmerkung:</i> Der Normalisierungsvorgang entspricht damit der Attributwertenormalisierung nach Abschnitt 3.3.3 der XML-Spezifikation.
<code>string translate (string1, string2, string3)</code>	Liefert die Zeichenkette <i>string1</i> wobei jedes Zeichen aus <i>string2</i> durch das Zeichen an derselben Position aus <i>string3</i> ersetzt wurde.

Tabelle 21: Boole'sche XPath-Funktionen

Funktionsprototyp	Funktionalität
<code>boolean boolean (object)</code>	Liefert die Boole'sche Repräsentation des übergebenen Arguments. Hierbei gilt: •Eine Zahl wird genau dann nach true konvertiert, wenn sie weder Null (unbeachtlich ihres Vorzeichens) noch eine nicht darstellbare Zahl (NaN) ist. •Eine Knotenmenge ergibt true, wenn sie nicht leer ist. •Eine Zeichenkette ergibt true, wenn sie nicht leer (d.h. Länge größer Null) ist. •Die Konvertierung anderer Typen ist typabhängig, und nicht durch den Standard festgelegt
<code>boolean not (boolean)</code>	Negiert das übergebene Argument

<code>boolean true()</code>	Liefert statisch den Wert true
<code>boolean false()</code>	Liefert statisch den Wert false
<code>boolean lang (string)</code>	Liefert true wenn der aktuelle Knoten ein xml:lang-Attribut gemäß der als Argument übergebenen Sprache besitzt

Tabelle 22: Zahlenorientierte XPath-Funktionen

Funktionsprototyp	Funktionalität
<code>number number (object?)</code>	Konvertiert ein Objekt in eine Zahl gemäß folgender Regeln: •Eine Zeichenkette wird in eine Fließkommazahl gemäß IEEE 754 konvertiert, wenn sie aus einem optionalen Leerzeichen, gefolgt durch ein optionales Minuszeichen, gefolgt von einem optionalen Leerzeichen und einer Ziffernfolge besteht. •Der Boole'sche Wert true wird zu 1, der Wert false zu 0 konvertiert. •Eine Knotenmenge wird zunächst in eine Zeichenkette übersetzt, und dann gemäß der oben definierten Regeln umgesetzt. •Die Konvertierung anderer Typen erfolgt typabhängig, und ist nicht durch den Standard geregelt. Wird kein Argument übergeben, so wird stattdessen der aktuelle Knoten als einziges Element einer Knotenmenge interpretiert.
<code>number sum (node-set)</code>	Liefert die Summe aller Elemente der übergebenen Knotenmenge, die zuvor in eine Zahl konvertiert werden.
<code>number floor (number)</code>	Liefert die größte ganze Zahl, die nicht größer als das Argument ist. <i>Anmerkung:</i> Entspricht dem Abschneiden beliebiger Nachkommastellen
<code>number ceiling (number1)</code>	Liefert die kleinste ganze Zahl, die nicht kleiner als das Argument ist. <i>Anmerkung:</i> Entspricht <code>floor(number1+0.999...)</code>
<code>number round (number)</code>	Liefert das Argument auf die nächste ganze Zahl gerundet. Gibt es zwei solche -- wie bei Nachkommastelle gleich 0.5 immer der Fall -- so wird die größere zurückgeliefert.

Für mathematische Berechnungen auf zahlenartigen Knoten stehen folgende Operatoren zur Verfügung.

Tabelle 23: Mathematische Operatoren

Operator	Funktionalität
+	Addition
-	Subtraktion als zweistelliger Operator. Der einstellige Operator - ist nicht spezifiziert, er liefert üblicherweise die negative Zahlendarstellung.
*	Multiplikation. Außer wenn innerhalb von XPath-Ausdrücken als Knotentest eingesetzt.
div	Division Achtung: Das Symbol / dient ausschließlich als Trennzeichen zur Separierung von Lokalisierungspfaden!
mod	Rest einer ganzzahligen Division

Ein umfangreiches Beispiel: Für das nachfolgende Beispiel wird das Projektverwaltungsdokument erweitert zu:

Beispiel 37: Erweiterte Projektverwaltung



```

(1)<?xml version="1.0" encoding="ISO-8859-15"?>
(2)<ProjektVerwaltung xmlns:xhtml="http://www.w3.org/1999/xhtml" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="L:\vorlesung\xml
\examples\projektverwaltung3.xsd">
(3)    <Person PersID="Pers01" mitarbeitInProjekt="Prj01">
(4)        <Vorname>Hans</Vorname>
(5)        <Nachname>Hinterhuber</Nachname>
(6)    </Person>
(7)    <Person PersID="Pers02" mitarbeitInProjekt="Prj02">
(8)        <Vorname>Franz</Vorname>
(9)        <Vorname>Xaver</Vorname>
(10)       <Nachname>Obermüller</Nachname>
(11)       <Qualifikationsprofil>
(12)           <xhtml:u>IT-Kompetenz</xhtml:u>
(13)           <xhtml:em>verschiedene</xhtml:em> Betriebssysteme und
(14)       <Leistungsstufe>professionelle</Leistungsstufe>
(15)           <xhtml:em>
(16)               <Qualifikation>Programmierung</Qualifikation>
(17)           </xhtml:em>
(18)       <xhtml:em>verschiedener Programmiersprachen
(19)           <xhtml:em>
(20)               <xhtml:u>
(21)                   <Qualifikation>Entwickler</Qualifikation>
(22)               </xhtml:u>
(23)           </xhtml:em> von 1988-1990
(24)       <xhtml:u>
(25)           <Qualifikation>Projektleiterfunktion</Qualifikation>
(26)       </xhtml:u>
(27)       von <xhtml:b>1990-93</xhtml:b> im X42-Projekt in Abteilung AB&C
(28)   </Qualifikationsprofil>
(29) </Person>
(30) <Person PersID="Pers03" mitarbeitInProjekt="Prj02">
(31)     <Vorname>Fritz</Vorname>
(32)     <Nachname>Meier</Nachname>
(33)     <Geburtsname value="Huber"/>
(34) </Person>
(35) <Projekt ID="Prj01" Projektleiter="Pers01" Mitarbeiter="Pers01"/>
(36) <Projekt ID="Prj02" Projektleiter="Pers02" Mitarbeiter="Pers03"/>
(37)</ProjektVerwaltung>

```

[Download des Beispiels](#)

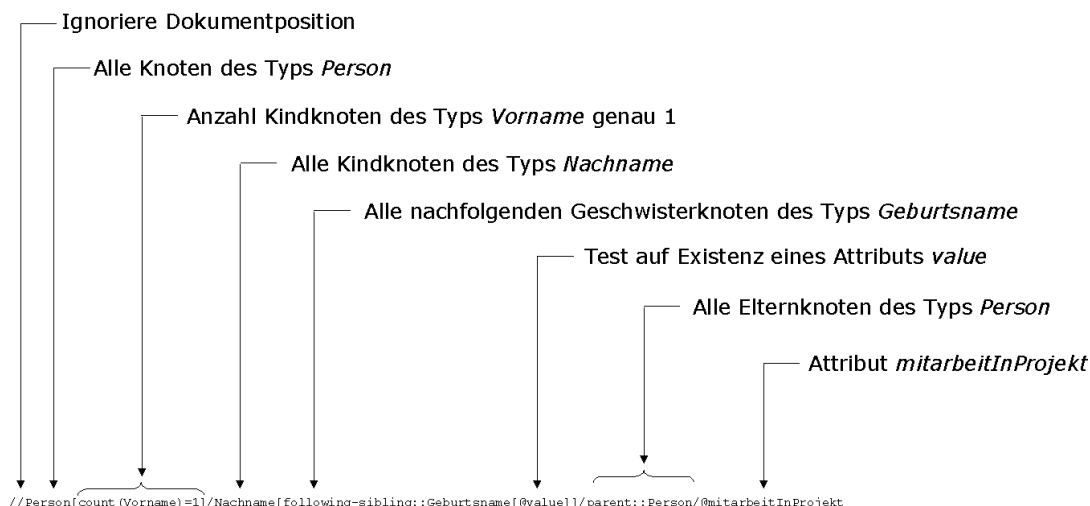


Abbildung 20: Auswertungsschritte (click on image to enlarge!)

Der XPath-Ausdruck der Abbildung 20 lokalisiert den Attributknoten des Inhalts Prj02.

Übung 2: Einige Übungen



Welches Ergebnis liefern folgende XPath-Ausdrücke?

- (a) `//Person[//child::Qualifikationsprofil]/Nachname`
- (b) `//Person[parent::ProjektVerwaltung]/Vorname[following-sibling::Vorname]`
- (c) `/ProjektVerwaltung/Person[attribute::PersID='Pers01']/Nachname`

Wie muß ein XPath-Ausdruck lauten, um folgendes zu selektieren?

- (d) Selektion aller Personen mit Nachnamen „Obermüller“.
- (e) Selektion aller Nachnamen von Personen die über mehr als eine Qualifikation verfügen.
- (f) Selektion der Nachnamen aller Projektleiter.

Anwendungsbeispiel: Integritätsbedingungen in XML-Schema

Über die Möglichkeiten der Datentypen hinausgehend bietet XML-Schema das Element `unique` zur Definition eindeutiger Wertbelegungen an. Hierbei wird auf die Lokatorsprache XPath zurückgegriffen um die abzu prüfenden Knoten innerhalb des Dokuments zu bezeichnen.

Die Syntax verwendet XPath-Ausdrücke eingeschränkter Mächtigkeit sowohl zur Festlegung des der Knotenmenge, auf die sich die Einschränkung bezieht (`selector`), als auch zur Angabe der eingeschränkten Knoten (`field`) selbst.

```
(1)<xsd:unique name="aName">
(2)    <xsd:selector xpath="aValidXPath"/>
(3)    <xsd:field xpath="aFieldStatement"/>
(4)    ...
(5)</xsd:unique>
```

Die Mächtigkeit der XPath-Ausdrücke ist dahingehend eingeschränkt, daß für das `selector`-Element ausschließlich Ausdrücke erlaubt sind, die Kindelemente des Knotens liefern, in dessen Kontext die durch `unique` formulierte Einschränkung angegeben wird. Als Konsequenz ist die Nutzung der verfügbaren XPath-Achsen auf diejenigen beschränkt, die Element-Knotenmengen zurückliefern.

Die Lokationsausdrücke in den -- möglicherweise mehrfach auftretenden -- `field`-Elementen werden relativ zum Pfad des `selector`-Knotens interpretiert. Hintereinandergesetzt muß der Pfad eines `selector`-Elements, gefolgt von einem Pfad eines `field`-Elements, einen gültigen Lokationsausdruck ergeben, der genau einen Knoten oder genau ein Attribut in der Ergebnismenge liefert. Sind mehrere `field`-Elemente zu einem `selector`-Element gegeben, so werden diese als durch logisches *und* verknüpft interpretiert. Mithin entspricht diese Semantik einem *concatenated primary key* aus den relationalen Datenbanken.

Das Beispiel zeigt die Nutzung des `unique`-Konstrukts zur Angabe der Eindeutigkeitsbedingung für das Attribut `PersID` des Elements `Person`.

Zunächst selektiert der Pfad `/Person` alle Knoten des gleichnamigen Typs; durch das `field`-Element wird die Eindeutigkeitsbedingung auf alle Attribut-Kindnoten des Typs `PersID` der Knoten in der selektierten Knotenmenge angewendet.

Die Semantik ist damit zur bisherigen ID-Typisierung identisch.

Beispiel 38: Unique-Einschränkung

```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsd:schema
(3)    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
(4)    elementFormDefault="qualified"
(5)    attributeFormDefault="unqualified">
(6)<xsd:element name="ProjektVerwaltung">
(7)    <xsd:complexType>
(8)        <xsd:sequence>
(9)            <xsd:element name="Person" type="PersonType"
maxOccurs="unbounded"/>
(10)            <xsd:element name="Projekt" type="ProjektType"
maxOccurs="unbounded"/>
(11)        </xsd:sequence>
(12)        <xsd:attribute name="version" type="xsd:string" fixed="1.0"/>
(13)    </xsd:complexType>
(14)    <xsd:unique name="uniquenessPersID">
(15)        <xsd:selector xpath="Person"/>
(16)        <xsd:field xpath="@PersID"/>
(17)    </xsd:unique>
(18)</xsd:element>
(19)
(20)<xsd:complexType name="PersonType">
(21)    <xsd:attribute name="PersID" type="xsd:token"/>
```



```

(22)</xsd:complexType>
(23)
(24)<xsd:complexType name="ProjektType"/>
(25)
(26)</xsd:schema>

```

[Download des Beispiels](#)

Das nächste Beispiel zeigt die Verwendung mehrerer field-Elemente zur Realisierung zusammengesetzter Schlüssel.

Hierzu wird die Kombination aus dem Inhalt des Nachnamen- und des Vornamen-Elements zusammen als eindeutig deklariert.

Überdies zeigt das Beispiel die Anwendung des Schlüsselmechanismus auf Elemente ohne Änderung der Basissyntax, abgesehen von der geänderten XPath-Achse.

Beispiel 39: Zusammengesetzter Schlüssel innerhalb eines unique-Elements



```

(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsd:schema
(3)    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
(4)    elementFormDefault="qualified"
(5)    attributeFormDefault="unqualified">
(6)<xsd:element name="ProjektVerwaltung">
(7)    <xsd:complexType>
(8)        <xsd:sequence>
(9)            <xsd:element name="Person" type="PersonType"
maxOccurs="unbounded"/>
(10)            <xsd:element name="Projekt" type="ProjektType"
maxOccurs="unbounded"/>
(11)        </xsd:sequence>
(12)        <xsd:attribute name="version" type="xsd:string" fixed="1.0"/>
(13)    </xsd:complexType>
(14)    <xsd:unique name="uniquenessPersID">
(15)        <xsd:selector xpath="Person"/>
(16)        <xsd:field xpath="Vorname"/>
(17)        <xsd:field xpath="Nachname"/>
(18)    </xsd:unique>
(19)</xsd:element>
(20)
(21)<xsd:complexType name="PersonType">
(22)    <xsd:sequence>
(23)        <xsd:element name="Vorname" type="xsd:token" minOccurs="1"
maxOccurs="unbounded"/>
(24)        <xsd:element name="Nachname" type="xsd:token" maxOccurs="1"/>
(25)    </xsd:sequence>
(26)</xsd:complexType>
(27)
(28)<xsd:complexType name="ProjektType"/>
(29)
(30)</xsd:schema>

```

[Download des Beispiels](#)

Zur Realisierung von wertdefinierenden Schlüsselbeziehungen bietet XML-Schema die Elemente [key](#) und [keyref](#) an. Sie werden verwendet um sicherzustellen, daß ein Element oder Attribut nur einen Wert annehmen darf, der bereits an anderer Stelle im Instanzdokument auftritt.

Hierzu lokalisiert key auf der Basis eines XPath-Ausdruckes eine Referenzmenge, während keyref diejenige Knotenmenge lokalisiert, in der ausschließlich Elemente der Referenzmenge enthalten sein dürfen.

Das Beispiel zeigt die Anwendung auf das Element ProjektVerwaltung. Der mit *projectKey* benannte Schlüssel definiert die Referenzmenge als das Ergebnis der Anfrage Projekt/@ID, worauf die *projectReference* Bezug nimmt.

Beispiel 40: Schlüsselbasierte Referenzierung


```

(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsd:schema
(3)    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
(4)    elementFormDefault="qualified"
(5)    attributeFormDefault="unqualified">
(6)    <xsd:element name="ProjektVerwaltung">
(7)        <xsd:complexType>
(8)            <xsd:sequence>
(9)                <xsd:element name="Person" type="PersonType"
maxOccurs="unbounded"/>
(10)                <xsd:element name="Projekt" type="ProjektType"
maxOccurs="unbounded"/>
(11)            </xsd:sequence>
(12)            <xsd:attribute name="version" type="xsd:string" fixed="1.0"/>
(13)        </xsd:complexType>
(14)
(15)        <xsd:key name="projectKey">
(16)            <xsd:selector xpath="Projekt"/>
(17)            <xsd:field xpath="@ID"/>
(18)        </xsd:key>
(19)        <xsd:keyref name="projectReference" refer="projectKey">
(20)            <xsd:selector xpath="Person"/>
(21)            <xsd:field xpath="@mitarbeitInProjekt"/>
(22)        </xsd:keyref>
(23)    </xsd:element>
(24)
(25)    <xsd:complexType name="PersonType">
(26)        <xsd:attribute name="mitarbeitInProjekt" type="xsd:token"/>
(27)    </xsd:complexType>
(28)    <xsd:complexType name="ProjektType">
(29)        <xsd:attribute name="ID" type="xsd:token"/>
(30)    </xsd:complexType>
(31)</xsd:schema>

```

[Download des Beispiels](#)



Web-Referenzen 7: Weiterführende Links

- [XPath Spezifikation](#)
- [Deutsche Übersetzung der XPath-Spezifikation](#)
- [XPath Visualisierer](#)

2.2 Java Database Connectivity (JDBC)

Motivation

Häufig besteht der Wunsch oder die Notwendigkeit auf bereits vorliegende Datenbestände, die durch ein Datenbankmanagementsystem (DBMS) verwaltet werden, in einer Applikationsprogrammiersprache zuzugreifen. Dabei soll die Anbindung der benötigten Datenquelle nicht problemspezifisch wieder und wieder neu entwickelt werden, sondern sollte sich auf ähnliche Datenanbindungsprobleme übertragen lassen.

Vor diesem Hintergrund liegt es nahe sich an den Typen der verfügbaren und kommerziell bedeutsamen DBMS zu orientieren und herstellersistenspezifische Entwicklungen außer Acht zu lassen. Gleichzeitig offenbaren sich hierbei Standardisierungsbemühungen wie die Sprache SQL zum Zugriff auf relationale DBMS als lohnenswerter Ansatz der Etablierung einer generischen und übertragbaren Schnittstelle.

Die Idee zur Schaffung einer solchen generischen Schnittstelle für den Zugriff auf relationale DBMS geht zurück auf eine Initiative der *SQL Access Group*, welche später in der Vereinigung mit der *X/Open Group* aufging, die zwischenzeitlich in *Open Group* umbenannt wurde. Das dort konzipierte programmiersprachenunabhängige *SQL Call Level Interface* (SQL/CLI) konnte sich dank der Umsetzung unter dem Namen Open Database Connectivity (ODBC) durch die Firma Microsoft und die parallel erfolgte internationale Normierung unter dem Titel *SQL/CLI* breit am Markt etablieren.

Die für die Programmiersprache Java adaptierte Variante des Zugriffs auf relationale DBMS wird durch SUN Microsystems unter dem Namen *Java Database Connectivity* (JDBC) propagiert und

stellt eine auf ODBC konzeptionell aufbauende und auf die spezifischen Bedürfnisse dieser Applikationsprogrammiersprache optimierte Untermenge des SQL/CLI-Standards dar.

Konzept und Grundidee

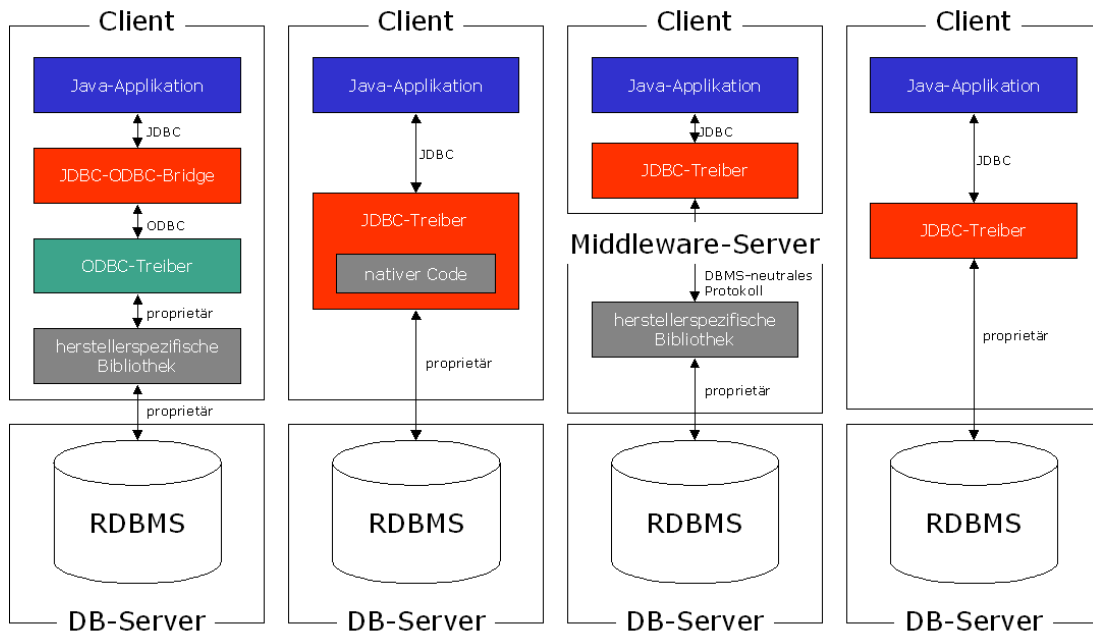
Von den Vorgängeransätzen übernommene Grundidee der Schnittstelle ist es den physischen Zugriff auf das Datenbankmanagementsystem durch eine von der Applikation spezialisierte wiederverwendbare Softwarekomponente, den sog. *JDBC-Treiber*, abzuwickeln.

Dieser Treiber vermittelt zwischen der Javaapplikation und dem verwendeten DBMS. Hierbei muß für jedes DBMS ein auf es abgestimmter JDBC-Treiber verwendet werden, da lediglich die Schnittstelle zur Applikation, nicht jedoch die zum DBMS, standardisiert ist.

Diesem Treiber obliegt die Abwicklung der gesamten Kommunikationsvorgänge mit dem DBMS. Er setzt jedoch selbst keine datenbankspezifischen Funktionalitäten, wie Syntax- oder Plausibilitätsprüfungen der übermittelten Kommandos um. Etwaige Fehlerprüfungen können, ebenso wie Anfrageoptimierungen, daher erst seitens des DBMS vorgenommen werden. Der Vorteil dieses Vorgehens liegt in der Generizität des JDBC-Treibers. Er kann ohne aufwendige Logikanteile als reine uninterpretierende Vermittlungsschicht zwischen Applikation und DBMS umgesetzt werden, wodurch schlanke Implementierungen ermöglicht werden.

Die [JDBC-Spezifikation](#) detailliert den Treiberbegriff zusätzlich hinsichtlich der gewählten technischen Umsetzung aus. So werden die vier in [Abbildung 3](#) dargestellten Treibertypen gemäß ihrer Charakteristika beschrieben und unterschieden.

Abbildung 3: JDBC-Treibertypen



Typ 1-Treiber Typ 2-Treiber Typ 3-Treiber Typ 4-Treiber
(click on image to enlarge!)

Die historisch älteste Variante bildet der **Typ 1 Treiber**. Strenggenommen verkörpert er selbst keinen Datenbanktreiber, sondern lediglich eine Umsetzungsschicht die einem existierenden ODBC-Treiber vorgeschaltet wird.

Die Abbildung belegt diesen Treibertyp daher mit dem Begriff *JDBC-ODBC-Bridge*, da er lediglich den Brückenschlag zwischen den beiden Standards vornimmt und sich in der konkreten Anwendung auf die Umsetzung der zwischen den beiden Protokollen beschränkt, ohne realen Zugriff auf die Datenbank zu erhalten.

Dieser ist dem ODBC-Treiber vorbehalten, der im allgemeinen Falle mit einer weiteren Umsetzungsstufe kommuniziert, welche die generischen ODBC-Aufrufe in konkrete DBMS-spezifische wandelt.

Während sowohl der JDBC-ODBC-Brückentreiber als auch der ODBC-Treiber selbst für verschiedene DBMS verwendet werden können, muß für jedes konkrete DBMS eine herstellerspezifische, d.h. an das verwendete DBMS angepaßte, Bibliothek vorliegen.

Für den Fall eines **Typ 2 Treibers** entfällt diese durch ODBC geschaffene zusätzliche Indirektionsstufe zugunsten der Adaption der Konversionskomponente, welcher die Wandlung der Aufrufe in das DBMS-native Protokoll obliegt, an das JDBC-Protokoll und ihrer Integration in den JDBC-Treiber selbst.

Die Natur der Kommunikation des Java-Anteils des Treibers mit den Nativen ist im Rahmen der durch die JDBC-Spezifikation gegebenen Definition nicht festgelegt.
Durch die Integration der DBMS-nativen Treiberanteile in den JDBC-Treiber muß dieser für jedes anzusprechende DBMS neu erstellt werden. Eine Wiederverwendung der JDBC-spezifischen Anteile die für die Clientkommunikation eingesetzt werden kann hierbei nicht erfolgen.

Der Fall der (partiellen) Konkretisierung dieser Kommunikationsbeziehung zu einem beliebigen *DBMS-neutralen Protokoll* wird durch einen **Typ 3 Treiber** aufgegriffen.
Hier wird die DBMS-spezifische Komponente (in der Abbildung grau dargestellt) als vom JDBC-Treiber separiertes Modul aufgefaßt, daß mit diesem mittels eines festgelegten neutralen Protokolls kommuniziert.

Durch diese Separierung, die auch durch Installation auf physisch getrennten Maschinen --- der DBMS-spezifische Anteil könnte beispielsweise auf einem Middleware-Server untergebracht werden --- fundiert werden kann, gelingt die Wiederverwendung des JDBC-Treiberanteils, der mit verschiedenen DBMS-spezifischen Bibliotheken über das gewählte Protokoll kommunizieren kann.

Der **Typ 4 Treiber** stellt die letzte durch die JDBC-Spezifikation vorgesehene Ausprägung dar. Er konzipiert eine vollständig in Java implementierte Zugriffsschicht, die in sich geschlossen ist. Sie besitzt daher lediglich die notwendige JDBC-Schnittstelle zur Kommunikation mit der Java-Applikation und eine DBMS-Spezifische zum Zugriff auf die Datenquelle.
Die Vorteile dieser Architekturvariante liegen in ihrer Portabilität und den geringen Installations und Wartungsaufwänden, die aus der Reduktion der Kommunikationsbeziehungen resultieren. So kann ein solcher Treiber durch einfache Integration in die Java-Applikation verwendet werden und bedarf keiner Installationen oder Modifikationen an der verwendeten Ausführungsumgebung.
Gleichzeitig offenbart sich diese Lösung jedoch als technisch aufwendig in der Umsetzung, sobald DBMS verschiedener Hersteller angesprochen werden sollen, da die JDBC-Anteile des Treibers nicht separat wiederverwendet werden können.

Hinsichtlich des Laufzeitverhaltens zeigt sich deutlich die Schwäche der Typ 1 Treiber, welche in der inhärent notwendigen Doppelkonversion (JDBC zu ODBC und ODBC zu nativem Aufruf) begründet liegt. Daher sind Treiber dieses Typs als Übergangserscheinung hin zu „echten“ JDBC-Treibern, d.h. Treibern der restlichen Typen, anzusehen und sollten in Produktivumgebungen nicht eingesetzt werden.

Die Vorteile der Typ 2 und 3 Treiber seitens der Ausführungsgeschwindigkeit liegen in den nativen Codeanteilen begründet, welche für das jeweilige verwendete DBMS optimiert werden können.
Zwar spricht der leichte Installations- und Administrationsaufwand eindeutig für Typ 4 Treiber, jedoch fallen diese in ihrer Leistungsfähigkeit durch die ausschließliche Verwendung der Programmiersprache Java teilweise deutlich hinter Treiber des Typs 2 und 3, mitunter sogar hinter solche des Typs 1, zurück. Sie verkörpern jedoch den aus konzeptioneller Sicht zu bevorzugenden Ansatz hinsichtlich Portabilität und Vergleichbarkeit der erzielten quantitativen Ergebnisse.
Typischerweise kommen im produktiven Einsatz jedoch Treiber der Typen 2 und 4 zum Einsatz, die entweder durch den Hersteller des DBMS mitgeliefert werden (Typ 2) oder auf der Basis publizierter Schnittstellen plattformunabhängig für genau ein spezifisches DBMS entwickelt wurden (Typ 4).

Generell formuliert das JDBC-Konzept auf dieser Ebene noch keine Einschränkung hinsichtlich der unterstützten DBMS-Typen und ist generell auf verschiedenste Datenquellen anwendbar. Durch die Struktur des API und die verfügbaren Treiber kristallisieren sich jedoch relationale DBMS als Hauptanwendungsgebiet dieser Zugriffsschnittstelle heraus.

Im folgenden wird die Verwendung des Typ 4 Treibers [Connector/J](#) im Zusammenspiel mit dem RDBMS *MySQL* betrachtet.

Die Beispiele basieren auf einer Demodatenbank, deren Struktur und Inhalte nachfolgend angegeben sind.

Die Tabelle *EMPLOYEE*

```

+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+
| FNAME   | MINIT | LNAME   | SSN       | BDATE     | ADDRESS                                     |
SEX | SALARY   | SUPERSSN | DNO |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+
| John     | B     | Smith   | 123456789 | 1965-01-09 | 731 Fondren, Houston, TX |
M   | 30000.00 | 333445555 | 5 |
| Franklin | T     | Wong    | 333445555 | 1955-12-08 | 638 Voss, Houston, TX   |
M   | 40000.00 | 888665555 | 5 |
| Joyce    | A     | English | 453453453 | 1972-07-31 | 5631 Rice, Houston, TX  |
F   | 25000.00 | 333445555 | 5 |

```

M	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX
M	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX
F	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX
M	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX
F	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX

Umsetzung in der Java-API

Das Klassendiagramm der [Abbildung 4](#) zeigt die zentralen Klassen des Paketes `java.sql`. Auffallend ist, daß alle Elemente des dargestellten Pakets -- abgesehen von den definierten Exceptionklassen -- als Schnittstellen ausgelegt sind. Durch diese Mimik wird die Organisation der JDBC-Schnittstelle deutlich. Die API legt lediglich das Verhalten hinsichtlich seiner Semantik und die Einzeloperationen durch Definition ihrer Parameter fest, die konkrete DBMS-spezifische Implementierung dieser Operationen wird durch den JDBC-Treiber bereitgestellt.

Zentrale Klasse der JDBC-API ist die Schnittstelle [Connection](#). Sie bildet die Kommunikationsverbindungen zum DBMS ab und bietet notwendige Verwaltungsoperationen. Hierunter fallen insbesondere auch die Aufrufe zur Transaktionssteuerung.

Die Schnittstelle [Statement](#) realisiert genau eine aus Javasicht atomare Datenbankaktion. Diese muß hierbei aus minimal einem Aufruf an das DBMS bestehen, kann aber eine Reihe separater Aufrufe zu einem *Batch* bündeln.

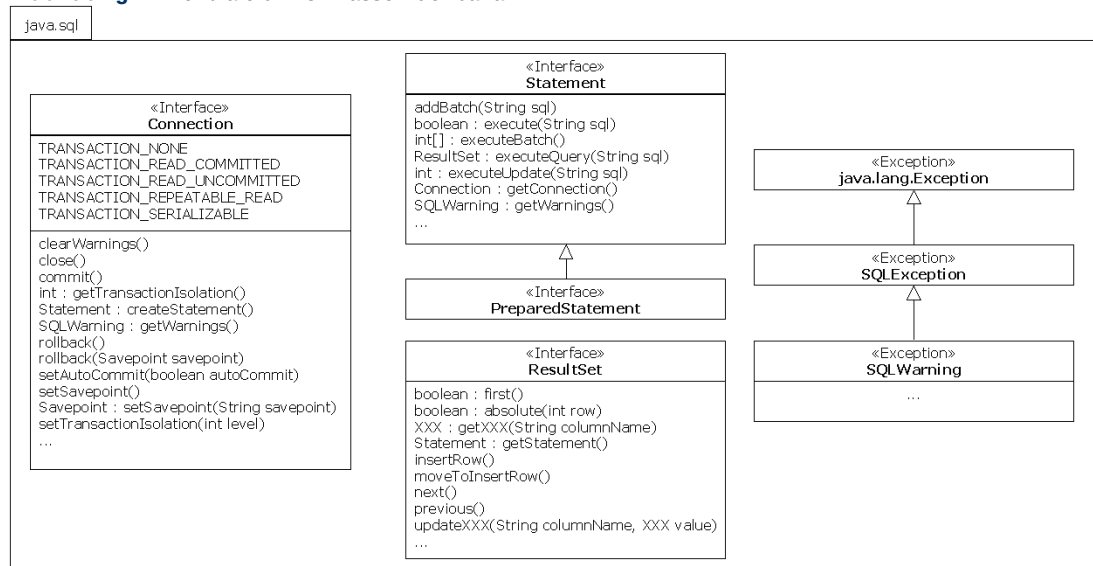
Als Sonderform sieht die API die Spezialisierung [PreparedStatement](#) vor, die es gestattet parametrisierte Anfragen zwischenspeichern, die nach Belegung der Parameterfelder an das DBMS übergeben werden. Hierdurch wird ein einfacher Mechanismus zur Wiederverwendung von DBMS-Aufrufen etabliert.

Liefert eine DBMS-Anfrage Ergebnistupel, wo werden diese konform zur Schnittstelle [ResultSet](#) verwaltet. Diese Schnittstelle erlaubt die lesende Traversierung der vom DBMS gelieferten Tupel ebenso wie ihre Aktualisierung im Hauptspeicher und das anschließende Zurückschreiben in die Datenbank.

Die in der Abbildung nur durch *getXXX* und *updateXXX* angedeuteten Operationen existieren in Ausprägungen für alle unterstützten Datentypen, wobei XXX den Namen des Typs bezeichnet.

Ferner definiert die API eine mit [SQLWarning](#) eine Ausnahme zur Behandlung auftretender Fehlersituationen sowie eine Reihe weiterer, in der [Abbildung 4](#) nicht dargestellter Klassen wie beispielsweise verschiedene Datentypen.

Abbildung 4: Zentrale JDBC-Klassen der Java-API



(click on image to enlarge!)

Zugriff auf die Datenbank

Beispiel 41 zeigt den Ablauf zur Aufnahme einer Verbindung mit der Datenbank jdbctest auf dem lokalen Rechner (localhost).

Zunächst muß die Klasse des gewählten JDBC-Treibers (im Beispiel `com.mysql.jdbc.Driver` vor ihrer Verwendung geladen werden. Dies geschieht durch den Aufruf der statischen Methode `forName` auf der Klasse `Class`.

Der zu ladende Treiber muß hierbei die JDBC-Schnittstellenklasse `Driver` implementieren um später durch die JDBC-API verwendet werden zu können.

Gleichzeitig mit dem dynamischen Ladevorgang erfolgt die Registrierung des Treibers beim JDBC-`DriverManager`, der die Verwaltung der geladenen DB-Treiber übernimmt.

Nach dem erfolgreichen Laden des Treibers wird durch den Aufruf von `getConnection` (Zeile 16) die Verbindung zur Datenbank hergestellt. Die anzusprechende Datenbank wird hierbei durch eine URI der Form `jdbc:mysql://DB-Server/DB-Name` repräsentiert (Zeile 17). Zusätzlich können ein zur Anmeldung am DB-System benötigter Benutzer (Zeile 18) und sein Paßwort (Zeile 19) übergeben werden.

Beispiel 41: Aufbau einer Datenbankverbindung



```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)
(5)public class JDBCConnect {
(6)    public static void main(String[] args) {
(7)        try {
(8)            Class.forName("com.mysql.jdbc.Driver");
(9)        } catch (ClassNotFoundException e) {
(10)            System.err.println("Driver class not found");
(11)            e.printStackTrace();
(12)        }
(13)        Connection con = null;
(14)        try {
(15)            con =
(16)                (Connection) DriverManager.getConnection(
(17)                    "jdbc:mysql://localhost/jdbctest/",
(18)                    "mario",
(19)                    "thePassword");
(20)        } catch (SQLException e1) {
(21)            System.err.println("Error establishing database connection");
(22)            e1.printStackTrace();
(23)        }
(24)    }
(25)
(26)}
```

[Download des Beispiels](#)

Zusätzlich stellen die Klassen `Driver` und `DriverManager` die Möglichkeit der Abfrage von verbindungsunabhängigen Verwaltungsinformationen zur Verfügung.

Beispiel 42: Ermittlung von Informationen über Treiber und Treibermanager

```
(1)import java.sql.Driver;
(2)import java.sql.DriverManager;
(3)import java.util.Enumeration;
(4)
(5)public class JDBCDriver {
(6)
(7)    public static void main(String[] args) {
(8)        try {
(9)            Class.forName("com.mysql.jdbc.Driver");
(10)        } catch (ClassNotFoundException e) {
(11)            System.err.println("Driver class not found");
(12)            e.printStackTrace();
(13)        }
(14)
(15)        System.out.println(
(16)            "DriverManager:\nlogin timeout=" + DriverManager.
getLoginTimeout());
(17)
(18)        Enumeration e = DriverManager.getDrivers();
```



```

(19)         while (e.hasMoreElements()) {
(20)             Driver drv = (Driver) e.nextElement();
(21)
(22)             System.out.println(
(23)                 "Driver="
(24)                     + drv.getClass().getName()
(25)                     + "\nmajor version="
(26)                     + drv.getMajorVersion()
(27)                     + "\nminor version="
(28)                     + drv.getMinorVersion()
(29)                     + "\nJDBC compliant="
(30)                     + drv.jdbcCompliant());
(31)         }
(32)
(33)     }
(34)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Beispiel 42 zeigt die Ermittlung des durch den DriverManager für alle durch ihn verwalteten Treiber global definierten Login Timouts, der angibt wie lange beim Anmeldevorgang an der Datenbank auf eine Rückmeldung gewartet wird.

Zusätzlich werden für alle verwalteten Treiber der Klassenname sowie Daten zur Version und zum Stand der JDBC-Unterstützung ermittelt und ausgegeben.

Der JDBC-Unterstützungsstand gibt an, ob ein gegebener Treiber die Konformitätstests der Firma SUN bestanden hat. Voraussetzung hierfür ist u.a. die vollständige Unterstützung des SQL 92-Standards (entry level).

Diese Interpretation von Spezifikationskonformität verwundert etwas, da alle JDBC-Treiber mit Ausnahme der inhärent DB-neutralen [Typ 1 Treiber](#) DBMS-spezifisch realisiert sind. Aus diesem Grunde bewertet der Konformitätstest vielmehr den Umsetzungsgrad des SQL-Standards in dem via JDBC genutzten DBMS als die Güte des JDBC-Treibers selbst.

Seit der JDBC-Schnittstellenversion 2 ist neben der „klassischen“ Zugriffsvariante auch eine auf dem [Java Naming and Directory Interface](#) (JNDI) basierende Zugriffsmethodik definiert, deren Verwendung --- abgesehen von der geänderten Mimik im Aufbau der DB-Verbindung --- identisch gestaltet ist.

Jedoch ist, wie in JNDI üblich, vor dem Zugriff ein benanntes Objekt beim JNDI-Dienst zu registrieren.

Im Falle von JDBC ist dies ein Objekt welches die Schnittstelle [DataSource](#) implementiert.

Der Code des Beispiels 43 zeigt die notwendigen Schritte zur Registrierung eines `MysqlDataSource`-Objekts, der durch den MySQL-JDBC-Treiber gelieferten Implementierung der Schnittstelle `DataSource`.

Beispiel 43: Ablage von Verbindungsinformation in einem JNDI-Verzeichnis

```

(1)import java.util.Hashtable;
(2)import javax.naming.Context;
(3)import javax.naming.InitialContext;
(4)import javax.naming.NamingException;
(5)import com.mysql.jdbc.jdbc2.optional.MysqlDataSource;
(6)
(7)public class JDBCConnect2Server {
(8)
(9)    public static void main(String[] args) {
(10)        Hashtable env = new Hashtable();
(11)        env.put(
(12)            Context.INITIAL_CONTEXT_FACTORY,
(13)            "com.sun.jndi.fscontext.RefFSContextFactory");
(14)        env.put(Context.PROVIDER_URL, "file:/tmp/registry");
(15)
(16)        MysqlDataSource ds = new MysqlDataSource();
(17)        ds.setDatabaseName("jdbctest");
(18)        Context ctx = null;
(19)        try {
(20)            ctx = new InitialContext(env);
(21)        } catch (NamingException ne) {
(22)            ne.printStackTrace();

```



```

(23)          }
(24)
(25)          try {
(26)              ctx.rebind("jdbc/mySrc", ds);
(27)          } catch (NamingException ne) {
(28)              ne.printStackTrace();
(29)          }
(30)      }
(31)}

```

[Download des Beispiels](#)

Entsprechend der modifizierten Ablage der Verwaltungsinformation ändert sich die Erzeugung der Datenbankverbindung beim Zugriff. Hier wird nun zunächst über einen Zugriff auf den JNDI-Verzeichnisdienst das benannte DataSource-Objekt (es trägt den Namen jdbc/mySrc ermittelt. Anschließend wird durch das dem Verzeichnisdienst entnommene DataSource-Objekt die Datenbankverbindung (d.h. das Connection-Objekt) erzeugt.

Alle weiteren Schritte zur Interaktion mit der Datenbank verlaufen dann identisch zur im Beispiel 41 gezeigten Verbindungsaufnahme.

Der Code des Beispiels 44 zeigt die notwendigen Schritte zur Ermittlung der Referenz auf das Objekt des Typs DataSource aus dem JNDI-Verzeichnis, sowie die Erzeugung des Connection-Objekts.

Beispiel 44: Verbindungsaufbau unter Nutzung von JNDI



```

(1)import java.sql.Connection;
(2)import java.sql.SQLException;
(3)import java.util.Hashtable;
(4)import javax.naming.Context;
(5)import javax.naming.InitialContext;
(6)import javax.naming.NamingException;
(7)import javax.sql.DataSource;
(8)
(9)public class JDBCConnect2 {
(10)    public static void main(String[] args) {
(11)        Hashtable env = new Hashtable();
(12)        env.put(
(13)            Context.INITIAL_CONTEXT_FACTORY,
(14)            "com.sun.jndi.fscontext.RefFSContextFactory");
(15)        env.put(Context.PROVIDER_URL, "file:/tmp/registry");
(16)        Context ctx = null;
(17)        try {
(18)            ctx = new InitialContext(env);
(19)        } catch (NamingException ne) {
(20)            ne.printStackTrace();
(21)        }
(22)        DataSource ds = null;
(23)        try {
(24)            ds = (DataSource) ctx.lookup("jdbc/mySrc");
(25)        } catch (NamingException ne) {
(26)            ne.printStackTrace();
(27)        }
(28)        Connection con = null;
(29)        try {
(30)            con = ds.getConnection("mario", "thePassword");
(31)        } catch (SQLException sqle) {
(32)            sqle.printStackTrace();
(33)        }
(34)    }
(35)}

```

[Download des Beispiels](#)

Auffallend ist die Ablage des Datenbanknamens im Verzeichnisdienst mittels des Methodenaufrufs setDatabaseName. Diese Verschiebung der Information wird durch die geänderte Mimik der Erzeugung des Connection-Objekts impliziert. So sieht die Implementierung dieser Methode für die Klasse DataSource keine Möglichkeit zur gleichzeitigen Übergabe von Anmeldenamen, Paßwort und Datenbank vor.

Vielmehrnoch ist es sogar möglich diese Daten allesamt innerhalb des JNDI-Verzeichnisdienstes abzulegen. (Für diesen Zweck stehen die Methoden setUser bzw. setPassword zur Verfügung.) Als

Konsequenz hiervon kann der Verbindungswunsch durch Aufruf der Methode `getConnection` ohne weitere Parameter erfüllt werden.

Diese Umsetzungsweise ist vor ihrer Realisierung hinsichtlich des damit eintretenden Verlustes an Sicherheit zu prüfen, da in ihrer Folge eine Datenbankverbindung allein durch Kenntnis des JNDI-residenten Namens des `DataSource`-Objektes erfolgen kann.

Generell wählen JDBC-Umsetzungen den Weg jede Ausprägung eines `Connection`-Objekts in eine physische Datenbankverbindung abzubilden. Dieses, durchaus der intuitiven Semantik der `Connection`-Klasse entsprechende Vorgehen kann jedoch in realen Applikationen, begründet in der Vielzahl der durch das DBMS zu verwaltenden Verbindungen, zu Zugriffsengpässen führen.

Aus diesem Grunde definiert die JDBC-Schnittstelle Operationen zur Zusammenfassung „gleichartiger“ Zugriffe. Hierzu zählen Zugriffe die unter derselben Nutzerkennung auf dieselbe Datenbank abgewickelt werden. Diese Zugriffsform tritt insbesondere bei Anwendungen auf, die über nur einen in der Datenbank eingetragenen Anwender verfügen und die gesamte Nutzerverwaltung datenbanktransparent applikationsseitig abwickeln.

Zur Optimierung von Zugriffen dieser Natur sieht die JDBC-Schnittstelle das sog. *Connection Pooling* vor, welches gleichartige Zugriffe bündelt.

Das Beispiel 45 zeigt eine Umsetzung:

Beispiel 45: Verbindungsaufbau unter Nutzung von Connection Pooling



```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import javax.sql.PooledConnection;
(4)import com.mysql.jdbc.Connection;
(5)import com.mysql.jdbc.jdbc2.optional.MysqlPooledConnection;
(6)
(7)public class JDBCConnection3 {
(8)    public static void main(String[] args) {
(9)        try {
(10)            Class.forName("com.mysql.jdbc.Driver");
(11)        } catch (ClassNotFoundException cnfe) {
(12)            System.err.println("Driver class not found");
(13)            cnfe.printStackTrace();
(14)        }
(15)        Connection con = null;
(16)        try {
(17)            con =
(18)                (Connection) DriverManager.getConnection(
(19)                    "jdbc:mysql://localhost/jdbctest/",
(20)                    "mario",
(21)                    "thePassword");
(22)        } catch (SQLException e1) {
(23)            System.err.println("Error establishing database connection");
(24)            e1.printStackTrace();
(25)        }
(26)
(27)        PooledConnection pc = new MysqlPooledConnection(con);
(28)
(29)        java.sql.Connection con1 = null;
(30)        try {
(31)            con1 = pc.getConnection();
(32)        } catch (SQLException sqle) {
(33)            sqle.printStackTrace();
(34)        }
(35)    }
(36)}
```

[Download des Beispiels](#)

Statt für jede gewünschte Datenbankverbindung ein zusätzliches Objekt des Type `Connection` zu erzeugen wird die erzeugte Verbindung zur Konstruktion eines Objektes, welches Konform zur Schnittstelle [PooledConnection](#) definiert ist, verwendet. Dieses verwaltet sort für die Verwaltung der DB-Verbindung und stellt dieselbe physische Verbindung verschiedenen Anfragern zur Verfügung.

Konsequenterweise wird daher eine neue Verbindung nicht mehr vom `DriverManager` angefordert, sondern durch die Methode `getConnection` der aus der Verwaltungsstruktur entnommenen `PooledConnection` beantragt.

Aufgrund der Unterstützung des SQL-Sprachumfanges, durch unveränderte textuelle Propagation

an das DBMS sind durch JDBC im Allgemeinen alle Facetten der Datenbanksprache nutzbar, sofern sie durch das verwendete DBMS Unterstützung finden. Hierunter fallen:

- Data Definition Language.
Zur Erzeugung eines Datenmodells.
- Data Manipulation Language.
Zur Modifikation der verwalteten Daten.
- Data Retrieval Language.
Zur Anfrage der in einer Datenbank gespeicherten Daten.
- Data Control Language.
Zur Festlegung und Kontrolle von Zugriffsberechtigungen.

JDBC reflektiert jedoch nicht diese Sprach(-sub-)klassen selbst in der API, sondern sieht vielmehr ausschließlich zwei Formen des Zugriffs vor. Solche die tabellenwerte Resultate liefern und solche, deren Ausführung lediglich primitivwertige Rückgabewerte liefert.

Primitivwertige Zugriffe

Primitivwertige Datenbankzugriffe liefern, abgesehen von Fehler- oder Warnmeldungen, lediglich die Anzahl der geänderten Tupel, falls zutreffend, oder 0 zurück.

Aus dieser Festlegung lassen sich diejenigen SQL-Anweisungstypen ableiten, welche als primitivwertiger Zugriff realisiert sind. Hierunter fallen alle Operationen der Datendefinition wie CREATE oder ALTER TABLE sowie alle Einfüge- (INSERT) Änderungs- (UPDATE) und Löschvorgänge (DELETE). Darüberhinaus alle Operationen zur Administration der Datenbank durch Rechtevergabe (GRANT, REVOKE).

Zugriffe dieser Art werden generell durch die Methode [executeUpdate](#), oder einer Abart davon, realisiert.

Beispiel 46: Erstellung einer neuen Tabelle

```
(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCCreateTable {
(7)    public static void main(String[] args) {
(8)        try {
(9)            Class.forName("com.mysql.jdbc.Driver");
(10)        } catch (ClassNotFoundException e) {
(11)            System.err.println("Driver class not found");
(12)            e.printStackTrace();
(13)        }
(14)        Connection con = null;
(15)
(16)        try {
(17)            con =
(18)                (Connection) DriverManager.getConnection(
(19)                    "jdbc:mysql://localhost/jdbctest/",
(20)                    "mario",
(21)                    "thePassword");
(22)        } catch (SQLException e1) {
(23)            System.err.println("Error establishing database connection");
(24)            e1.printStackTrace();
(25)        }
(26)
(27)        Statement stmt = null;
(28)        try {
(29)            stmt = (Statement) con.createStatement();
(30)        } catch (SQLException e2) {
(31)            System.err.println("Error creating SQL-Statement");
(32)            e2.printStackTrace();
(33)        }
(34)        String createTab = new String("CREATE TABLE EMPLOYEE(" +
(35)            "FNAME VARCHAR(10) NOT NULL," +
(36)            "MINIT VARCHAR(1)," +
(37)            "LNAME VARCHAR(10) NOT NULL," +
(38)            "SSN INTEGER(9) NOT NULL," +
(39)            "BDATE DATE," +
(40)            "ADDRESS VARCHAR(30)," +
(41)            "SEX ENUM('M','F')," +
```




```

(42)                "SALARY REAL(7,2) UNSIGNED," +
(43)                "SUPERSSN INTEGER(9)," +
(44)                "DNO INTEGER(1));";
(45)                try {
(46)                    System.out.println("result="+stmt.executeUpdate(createTab));
(47)                } catch (SQLException e3) {
(48)                    System.err.println("Error creating table EMPLOYEE");
(49)                    e3.printStackTrace();
(50)                }
(51)            }
(52)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Beispiel 46 zeigt die notwendigen Schritte zur Erstellung der [Tabelle EMPLOYEE](#) in der Datenbank.

Nach dem (üblichen) Verbindungsaufbau (Zeile 8-24) wird in Zeile 26 eine Variable des Typs [Statement](#) deklariert. Auch bei Statement handelt es sich um eine durch die JDBC-API vordefinierte Schnittstelle die als Bestandteil des JDBC-Treibers durch von einer Klasse implementiert wird.

Ausgehend von der etablierten Datenbankverbindung wird durch Aufruf der Methode [createStatement](#) eine konkrete Ausprägung konform zur Statement-Schnittstelle erzeugt (Zeile 28).

Der Aufruf von [executeUpdate](#) übergibt das als Zeichenkette abgelegte SQL-Kommando an die Datenbank zur Ausführung.

Da durch CREATE TABLE keine Tupeländerungen vorgenommen werden ist das Resultat des Aufrufs der Rückgabewert 0.

Beispiel 47 zeigt mit dem ALTER TABLE-Kommando eine weitere Anwendung der executeUpdate-Methode.

Auch in diesem Falle wird als Resultat 0 geliefert, da die Definition des Primärschlüssels keine Änderungen an den verwalteten Datensätzen vornimmt.

Beispiel 47: Modifikation der Tabellendefinition

```

(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCAlterTable {
(7)    public static void main(String[] args) {
(8)        try {
(9)            Class.forName("com.mysql.jdbc.Driver");
(10)        } catch (ClassNotFoundException e) {
(11)            System.err.println("Driver class not found");
(12)            e.printStackTrace();
(13)        }
(14)        Connection con = null;
(15)
(16)        try {
(17)            con =
(18)                (Connection) DriverManager.getConnection(
(19)                    "jdbc:mysql://localhost/jdbctest/",
(20)                    "mario",
(21)                    "thePassword");
(22)        } catch (SQLException e1) {
(23)            System.err.println("Error establishing database connection");
(24)            e1.printStackTrace();
(25)        }
(26)
(27)        Statement stmt = null;
(28)        try {
(29)            stmt = (Statement) con.createStatement();
(30)        } catch (SQLException e2) {
(31)            System.err.println("Error creating SQL-Statement");
(32)            e2.printStackTrace();
(33)        }
(34)        String createTab =

```



```

(35)         new String("ALTER TABLE EMPLOYEE ADD PRIMARY KEY (SSN);");
(36)         try {
(37)             System.out.println("result=" + stmt.executeUpdate
(createTab));
(38)         } catch (SQLException e3) {
(39)             System.err.println("Error altering table EMPLOYEE");
(40)             e3.printStackTrace();
(41)         }
(42)     }
(43)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Beispiel 48: Einfügen von Werten

```

(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCInsert1 {
(7)    public static void main(String[] args) {
(8)        try {
(9)            Class.forName("com.mysql.jdbc.Driver");
(10)        } catch (ClassNotFoundException e) {
(11)            System.err.println("Driver class not found");
(12)            e.printStackTrace();
(13)        }
(14)        Connection con = null;
(15)
(16)        try {
(17)            con =
(18)                (Connection) DriverManager.getConnection(
(19)                    "jdbc:mysql://localhost/jdbctest/",
(20)                    "mario",
(21)                    "thePassword");
(22)        } catch (SQLException e1) {
(23)            System.err.println("Error establishing database connection");
(24)            e1.printStackTrace();
(25)        }
(26)
(27)        Statement stmt = null;
(28)        try {
(29)            stmt = (Statement) con.createStatement();
(30)        } catch (SQLException e2) {
(31)            System.err.println("Error creating SQL-Statement");
(32)            e2.printStackTrace();
(33)        }
(34)
(35)        try {
(36)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('John', 'B', 'Smith', 123456789, '1965-01-09', '731 Fondren,
Houston, TX', 'M', 30000, 333445555, 5);"));
(37)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Franklin', 'T', 'Wong', 333445555, '1955-12-08', '638 Voss,
Houston, TX', 'M', 40000, 888665555, 5);"));
(38)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Alicia', 'J', 'Zelaya', 999887777, '1968-07-19', '3321 Castle,
Spring, TX', 'F', 25000, 987654321, 4);"));
(39)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Jennifer', 'S', 'Wallace', 987654321, '1941-06-20', '291
Berry, Bellaire, TX', 'F', 43000, 888665555, 4);"));
(40)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Ramesh', 'K', 'Narayan', 666884444, '1962-09-15', '975 Fire
Oak, Humble, TX', 'M', 38000, 333445555, 5);"));
(41)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Joyce', 'A', 'English', 453453453, '1972-07-31', '5631 Rice,
Houston, TX', 'F', 25000, 333445555, 5);"));
(42)            System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('Ahmad', 'V', 'Jabbar', 987987987, '1969-03-29', '980 Dallas,

```



```

Houston, TX', 'M', 25000, 987654321, 4);"));
(43)          System.out.println("result=" + stmt.executeUpdate("INSERT
INTO EMPLOYEE VALUES('James', 'E', 'Borg', 888665555, '1937-11-10', '450 Stone,
Houston, TX', 'M', 55000, null, 1);"));
(44)          } catch (SQLException e3) {
(45)              System.err.println("Error inserting values into table
EMPLOYEE");
(46)              e3.printStackTrace();
(47)          }
(48)      }
(49)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Beispiel 48 zeigt den Einfügevorgang von acht Werten in die durch die vorangegangenen Beispiele erzeugte Tabelle EMPLOYEE.

Jeder der Einfügevorgänge der Zeilen 36-43 führt im Rahmen einer separaten Datenbankkommunikation sequentiell genau einen Einfügevorgang durch, was durch den Rückgabewert 1 dokumentiert wird.

Zwar ist dieses Verfahren praktikabel und erzielt die angestrebten Resultate, jedoch ist es unter Zeiteffizienzgesichtspunkten inadäquat, da sich Einfüge- und Kommunikationsvorgänge zahlenmäßig entsprechen.

Aus diesem Grunde bietet die Schnittstelle [Statement](#) die Möglichkeit zur Bündelung einzelner SQL-Aufrufe in einem sog. *Batch* an.

Beispiel 49 zeigt die entsprechende Umgestaltung des vorangegangenen Beispiels.

Beispiel 49: Einfügen von Werten mittels eines Batches

```

(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCInsert2 {
(7)    public static void main(String[] args) {
(8)        try {
(9)            Class.forName("com.mysql.jdbc.Driver");
(10)        } catch (ClassNotFoundException e) {
(11)            System.err.println("Driver class not found");
(12)            e.printStackTrace();
(13)        }
(14)        Connection con = null;
(15)
(16)        try {
(17)            con =
(18)                (Connection) DriverManager.getConnection(
(19)                    "jdbc:mysql://localhost/jdbctest/",
(20)                    "mario",
(21)                    "thePassword");
(22)        } catch (SQLException e1) {
(23)            System.err.println("Error establishing database connection");
(24)            e1.printStackTrace();
(25)        }
(26)
(27)        Statement stmt = null;
(28)        try {
(29)            stmt = (Statement) con.createStatement();
(30)        } catch (SQLException e2) {
(31)            System.err.println("Error creating SQL-Statement");
(32)            e2.printStackTrace();
(33)        }
(34)
(35)        try {
(36)            stmt.addBatch("INSERT INTO EMPLOYEE VALUES('John', 'B',
'Smith', 123456789, '1965-01-09', '731 Fondren, Houston, TX', 'M', 30000, 333445555,
5);");
(37)            stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Franklin', 'T',

```



```

'Wong', 333445555, '1955-12-08', '638 Voss, Houston, TX', 'M', 40000, 888665555,
5);");
(38)          stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Alicia', 'J',
'Zelaya', 999887777, '1968-07-19', '3321 Castle, Spring, TX', 'F', 25000, 987654321,
4);");
(39)          stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Jennifer', 'S',
'Wallace', 987654321, '1941-06-20', '291 Berry, Bellaire, TX', 'F', 43000,
888665555, 4);");
(40)          stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Ramesh', 'K',
'Narayan', 666884444, '1962-09-15', '975 Fire Oak, Humble, TX', 'M', 38000,
333445555, 5);");
(41)          stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Joyce', 'A',
'English', 453453453, '1972-07-31', '5631 Rice, Houston, TX', 'F', 25000, 333445555,
5);");
(42)          stmt.addBatch("INSERT INTO EMPLOYEE VALUES('Ahmad', 'V',
'Jabbar', 987987987, '1969-03-29', '980 Dallas, Houston, TX', 'M', 25000, 987654321,
4);");
(43)          stmt.addBatch("INSERT INTO EMPLOYEE VALUES('James', 'E',
'Borg', 888665555, '1937-11-10', '450 Stone, Houston, TX', 'M', 55000, null, 1);");
(44)          int[] insertCounts = stmt.executeBatch();
(45)          } catch (SQLException e3) {
(46)              System.err.println("Error inserting values into table
EMPLOYEE");
(47)              e3.printStackTrace();
(48)          }
(49)      }
(50)}

```

[Download des Beispiels](#)

Statt der Einzelübergabe der SQL INSERT-Anweisungen werden diese nun (in Zeile 36-43) in einem Batch gesammelt. Hierzu werden die SQL-Zeichenketten durch den Aufruf [addBatch](#) innerhalb des Statement-Objekts abgelegt und durch Aufruf der Methode [executeBatch](#) gesammelt an das DBMS übergeben.

Statt der Einzelresultate wird durch diese Aufrufvariante ein Array geliefert, der die Einzelrückgabewerte der als Batch übergebenen Aufrufe versammelt.

Dies verdeutlicht nochmals das nachfolgende Beispiel. In ihm wird zunächst mittels ALTER TABLE eine neue Tabellenspalte zur Aufnahme des Wochentages der Geburt erstellt und anschließend durch SQL UPDATE-Anweisungen die benötigten Daten aus dem vorhandenen Geburtsdatum ermittelt.

Auch dieses Beispiel bedient sich zur Performancebeschleunigung der Möglichkeiten des Batchaufrufes.

Beispiel 50: Aktualisieren von Tabellendefinitionen und Werten

```

(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.Statement;
(5)
(6)public class JDBCUpdate1 {
(7)    public static void main(String[] args) {
(8)        try {
(9)            Class.forName("com.mysql.jdbc.Driver");
(10)        } catch (ClassNotFoundException e) {
(11)            System.err.println("Driver class not found");
(12)            e.printStackTrace();
(13)        }
(14)        Connection con = null;
(15)
(16)        try {
(17)            con =
(18)                (Connection) DriverManager.getConnection(
(19)                    "jdbc:mysql://localhost/jdbctest/",
(20)                    "mario",
(21)                    "thePassword");
(22)        } catch (SQLException e1) {
(23)            System.err.println("Error establishing database connection");
(24)            e1.printStackTrace();
(25)        }

```



```

(26)
(27)         Statement stmt = null;
(28)         try {
(29)             stmt = (Statement) con.createStatement();
(30)         } catch (SQLException e2) {
(31)             System.err.println("Error creating SQL-Statement");
(32)             e2.printStackTrace();
(33)         }
(34)
(35)         try {
(36)             stmt.addBatch("ALTER TABLE EMPLOYEE ADD BDAY VARCHAR(10);");
(37)             stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Sunday' WHERE
DAYOFWEEK(BDATE)=1;");
(38)             stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Monday' WHERE
DAYOFWEEK(BDATE)=2;");
(39)             stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Tuesday' WHERE
DAYOFWEEK(BDATE)=3;");
(40)             stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Wednesday' WHERE
DAYOFWEEK(BDATE)=4;");
(41)             stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Thursday' WHERE
DAYOFWEEK(BDATE)=5;");
(42)             stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Friday' WHERE
DAYOFWEEK(BDATE)=6;");
(43)             stmt.addBatch("UPDATE EMPLOYEE SET BDAY='Saturday' WHERE
DAYOFWEEK(BDATE)=7;");
(44)             int[] result = stmt.executeBatch();
(45)             for (int i=0; i<result.length;i++){
(46)                 System.out.println("Statement No "+i+" changed
"+result[i]+" rows");
(47)             }
(48)         } catch (SQLException e3) {
(49)             System.err.println("Error inserting values into table
EMPLOYEE");
(50)             e3.printStackTrace();
(51)         }
(52)     }
(53)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Die Ausführung liefert als Resultat:

```

Statement No 0 changed 8 rows
Statement No 1 changed 0 rows
Statement No 2 changed 1 rows
Statement No 3 changed 0 rows
Statement No 4 changed 1 rows
Statement No 5 changed 1 rows
Statement No 6 changed 2 rows
Statement No 7 changed 3 rows

```

So werden durch den ALTER TABLE-Aufruf (Indexnummer 0) alle acht Tupel der Tabelle modifiziert, während die nachfolgenden Aufrufe nur Teilmengen davon verändern.

Die nähere Betrachtung der Zeilen 37-43 des Quellcodes von Beispiel 50 zeigt sich, daß diese im Kern denselben Vorgang ausführen, nur jeweils mit variierenden Parametern.

Zur Behandlung von Fällen dieser Problemstellung definiert die JDBC-API die Schnittstelle [PreparedStatement](#) als Spezialisierung von Statement.

Diese Schnittstelle gestattet es Anweisungen die später an die Datenbank übermittelt werden sollen mit Platzhaltern zu versehen und diese vor der Übermittlung mit Werten zu befüllen.

Beispiel 51 zeigt die entsprechende Modifikation des vorangegangenen Beispiels.

Beispiel 51: Aktualisieren von Tabellendefinitionen und Werten

```

(1)import java.sql.DriverManager;
(2)import java.sql.SQLException;
(3)import com.mysql.jdbc.Connection;
(4)import com.mysql.jdbc.PreparedStatement;
(5)import com.mysql.jdbc.Statement;
(6)
(7)public class JDBCUpdate2 {
(8)    public static void main(String[] args) {
(9)        try {
(10)            Class.forName("com.mysql.jdbc.Driver");
(11)        } catch (ClassNotFoundException e) {
(12)            System.err.println("Driver class not found");
(13)            e.printStackTrace();
(14)        }
(15)        Connection con = null;
(16)
(17)        try {
(18)            con =
(19)                (Connection) DriverManager.getConnection(
(20)                    "jdbc:mysql://localhost/jdbctest/",
(21)                    "mario",
(22)                    "thePassword");
(23)        } catch (SQLException e1) {
(24)            System.err.println("Error establishing database connection");
(25)            e1.printStackTrace();
(26)        }
(27)
(28)        Statement stmt = null;
(29)        PreparedStatement pstmt = null;
(30)        try {
(31)            stmt = (Statement) con.createStatement();
(32)            pstmt = (PreparedStatement) con.prepareStatement("UPDATE
EMPLOYEE SET BDAY=? WHERE DAYOFWEEK(BDATE)=?;");
(33)
(34)        } catch (SQLException e2) {
(35)            System.err.println("Error creating SQL-Statement");
(36)            e2.printStackTrace();
(37)        }
(38)
(39)        try {
(40)            String[] days=
{"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"} ;
(41)            stmt.addBatch("ALTER TABLE EMPLOYEE ADD BDAY VARCHAR(10);");
(42)            for (int i=1; i<8;i++){
(43)                pstmt.setString(1,days[i-1]);
(44)                pstmt.setInt(2,i);
(45)                pstmt.addBatch();
(46)            }
(47)            int[] result = stmt.executeBatch();
(48)            for (int i=0; i<result.length;i++){
(49)                System.out.println("Statement No "+i+" changed
"+result[i]+" rows");
(50)            }
(51)        } catch (SQLException e3) {
(52)            System.err.println("Error inserting values into table
EMPLOYEE");
(53)            e3.printStackTrace();
(54)        }
(55)    }
(56)}

```



[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Im Beispiel wird neben dem Objekt des Typs Statement zusätzlich eines des Typs PreparedStatement erzeugt (Zeile 32).

Die dem Konstruktor übergebene Anweisung enthält als Sonderzeichen zur Markierung der Platzhalter das Fragezeichen (?).

Die Wochentage werde in Zeile 40, des vereinfachten Zugriffs wegen, als Array definiert.

In den Zeilen 42 mit 46 werden die benötigten SQL-UPDATE-Anweisungen dynamisch aus durch

Einsetzen der geeigneten Werte in den vorpräparierten Änderungsausdruck erzeugt und einem eigenen Batch zugeordnet. Der Einsetzungsvorgang der benötigten Werte geschieht durch die Methoden [setString](#) für zeichenkettenartige bzw. [setInt](#) für den ganzzahlige Parameter. Den Methoden wird jeweils die Position des Parameters, gezählt ab 1 sowie die zu wählende Wertbelegung übermittelt.

Zur Ausführung müssen beide Batches getrennt angefordert werden.

Tabellenwertige Zugriffe

Die in der Praxis quantitativ bedeutendste Klasse von Datenbankzugriffen dürfte zweifellos auf die lesende Ermittlung von bestehenden Daten darstellen, kurzum alle Spielarten der SQL SELECT-Anweisung.

Für Anfragen an die Datenbank steht prinzipiell der gesamte durch das DBMS unterstützte SQL-Umfang zur Verfügung.

Anfragen werden im Gegensatz zu den bisher betrachteten lesenden Zugriffen nicht als primivwerte Methoden realisiert, sondern liefern als Resultat immer eine Tabelle zurück.

Diese wird durch den API-Typ [ResultSet](#) dargestellt.

Zusätzlich werden Anfragen durch die Methode [executeQuery](#) ausgeführt.

Das Beispiel 52 zeigt die generische Extraktion von DB-Daten und den Zugriff auf Metadaten. Die aus der Datenbank gelesenen Ergebnistupel werden im durch rs benannten ResultSet abgelegt (Zeile 39). Die Resultatmenge wird mithilfe eines *Cursors* (Datensatzzeiger) traversiert. Hierzu wird der initial auf eine Ausgangsstellung vor dem ersten empfangenen Tupel positionierte Cursor durch Aufruf der Methode `next` solange weitergerückt, bis der letzte Datensatz verarbeitet wurde.

Der Aufruf der Methode [getMetaData](#) liefert deskriptive Metadaten wie Spaltenzahl sowie deren Bezeichner und Typen für die erstellte Resultattupelmenge.

In Zeile 43 werden diese Metadaten verwendet um die Spaltennamen der extrahierten Attribute anzuzeigen.

Zeile 47-52 liest die einzelnen Werte jedes Tupels mittels [getObject](#) aus und stellt sie am Bildschirm dar.

Beispiel 52: Auslesen von Daten und Metadaten

```
(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.ResultSetMetaData;
(4)import java.sql.SQLException;
(5)
(6)import com.mysql.jdbc.Connection;
(7)import com.mysql.jdbc.Statement;
(8)
(9)public class JDBCSelect1 {
(10)    public static void main(String[] args) {
(11)        try {
(12)            Class.forName("com.mysql.jdbc.Driver");
(13)        } catch (ClassNotFoundException e) {
(14)            System.err.println("Driver class not found");
(15)            e.printStackTrace();
(16)        }
(17)        Connection con = null;
(18)
(19)        try {
(20)            con =
(21)                (Connection) DriverManager.getConnection(
(22)                    "jdbc:mysql://localhost/jdbctest/",
(23)                    "mario",
(24)                    "thePassword");
(25)        } catch (SQLException e1) {
(26)            System.err.println("Error establishing database connection");
(27)            e1.printStackTrace();
(28)        }
(29)
(30)        Statement stmt = null;
(31)        try {
(32)            stmt = (Statement) con.createStatement();
(33)        } catch (SQLException e2) {
(34)            System.err.println("Error creating SQL-Statement");
```




```

(35)                e2.printStackTrace();
(36)            }
(37)
(38)            try {
(39)                ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(40)                ResultSetMetaData rsmd = rs.getMetaData();
(41)                int noColumns = rsmd.getColumnCount();
(42)                for (int i = 1; i < noColumns; i++) {
(43)                    System.out.print(rsmd.getColumnLabel(i) + "\t");
(44)                }
(45)                System.out.println();
(46)
(47)                while (rs.isLast() == false) {
(48)                    rs.next();
(49)                    for (int i = 1; i < noColumns; i++) {
(50)                        System.out.print( rs.getObject(i)+"\t" );
(51)                    }
(52)                    System.out.println();
(53)                }
(54)
(55)            } catch (SQLException e3) {
(56)                System.err.println("Error selecting values from table
EMPLOYEE");
(57)                e3.printStackTrace();
(58)            }
(59)        }
(60)    }

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Neben im Beispiel 52 gezeigten Verarbeitung in exakter der Ablagereihenfolge der Datenbank kann auch durch Definition eines Cursors die Traversierung in inverser Ablagerichtung erreicht werden. Das nachfolgende Beispiel das entsprechende Vorgehen durch anfängliche Positionierung des Cursors ans Ende der empfangenen Daten (d.h. nach dem letzten Datensatz) und anschließendes schrittweises Rückpositionieren durch Aufruf der Methode `previous`.

Beispiel 53: Auslesen von Daten in invertierter Reihenfolge

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import com.mysql.jdbc.Connection;
(5)import com.mysql.jdbc.Statement;
(6)
(7)public class JDBCSelect5 {
(8)    public static void main(String[] args) {
(9)        try {
(10)            Class.forName("com.mysql.jdbc.Driver");
(11)        } catch (ClassNotFoundException e) {
(12)            System.err.println("Driver class not found");
(13)            e.printStackTrace();
(14)        }
(15)        Connection con = null;
(16)
(17)        try {
(18)            con =
(19)                (Connection) DriverManager.getConnection(
(20)                    "jdbc:mysql://localhost/jdbctest/",
(21)                    "mario",
(22)                    "thePassword");
(23)        } catch (SQLException sqle) {
(24)            System.err.println("Error establishing database connection");
(25)            sqle.printStackTrace();
(26)        }
(27)
(28)        Statement stmt = null;
(29)        try {
(30)            stmt =
(31)                (Statement) con.createStatement();
(32)        } catch (SQLException e2) {
(33)            System.err.println("Error creating SQL-Statement");

```




```

(34)                e2.printStackTrace();
(35)            }
(36)
(37)            try {
(38)                ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(39)                rs.afterLast();
(40)                while (rs.previous()){
(41)                    System.out.println(rs.getString("FNAME"));
(42)                }
(43)            } catch (SQLException sqle) {
(44)                System.err.println("Error selecting values from table
EMPLOYEE");
(45)                sqle.printStackTrace();
(46)            }
(47)        }
(48)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Ferner kann der Cursor wahlfrei auf eine beliebige Position der Ergebnisrelation gesetzt werden. Das nachfolgende Beispiel zeigt dies. Ferner illustriert es das Vorgehen zur Größenermittlung des resultierenden ResultSets durch das Aufrufpaar `last` und `getRow`, welches zunächst den Cursor auf den letzten aus der Datenbank extrahierten Datensatz positioniert und anschließend dessen Nummer liefert.

Beispiel 54: Auslesen von Daten in wahlfreier Reihenfolge

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import com.mysql.jdbc.Connection;
(5)import com.mysql.jdbc.Statement;
(6)
(7)public class JDBCSelect6 {
(8)    public static void main(String[] args) {
(9)        try {
(10)            Class.forName("com.mysql.jdbc.Driver");
(11)        } catch (ClassNotFoundException e) {
(12)            System.err.println("Driver class not found");
(13)            e.printStackTrace();
(14)        }
(15)        Connection con = null;
(16)
(17)        try {
(18)            con =
(19)                (Connection) DriverManager.getConnection(
(20)                    "jdbc:mysql://localhost/jdbctest/",
(21)                    "mario",
(22)                    "thePassword");
(23)        } catch (SQLException sqle) {
(24)            System.err.println("Error establishing database connection");
(25)            sqle.printStackTrace();
(26)        }
(27)
(28)        Statement stmt = null;
(29)        try {
(30)            stmt = (Statement) con.createStatement();
(31)        } catch (SQLException e2) {
(32)            System.err.println("Error creating SQL-Statement");
(33)            e2.printStackTrace();
(34)        }
(35)
(36)        try {
(37)            int position = 0;
(38)            ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(39)            rs.last();
(40)            int size = rs.getRow();
(41)            for (int i = 0; i < size; i++) {
(42)                position = (position + 3) % size;
(43)                rs.absolute(position + 1);

```



```

(44)                System.out.println(
(45)                "position=" + (position + 1) + ": " + rs.
getString("FNAME"));
(46)                }
(47)            } catch (SQLException sqle) {
(48)                System.err.println("Error selecting values from table
EMPLOYEE");
(49)                sqle.printStackTrace();
(50)            }
(51)        }
(52)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Wird der benötigte ResultSet geeignet (d.h. mit den Parameter CONCUR_UPDATABLE) (siehe Zeile 49) initialisiert so können Änderungen, die im Hauptspeicher durch die JDBC-API durchgeführt werden in die Datenbank persistiert werden.

Beispiel 55 zeigt dies exemplarisch für den Einfügevorgang eines neuen Tupels.

Die Voraussetzungen für Einfüge- und Aktualisierungsvorgänge entsprechen denen von *updatable* views, d.h. die Daten dürfen nur aus genau einer Tabelle entnommen sein und müssen den Primärschlüssel enthalten.

Beispiel 55: Auslesen und Einfügen von Daten

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.ResultSetMetaData;
(4)import java.sql.SQLException;
(5)import java.sql.Statement;
(6)
(7)import com.mysql.jdbc.Connection;
(8)
(9)public class JDBCSelect2 {
(10)    private static void printResultSet(ResultSet rs) throws SQLException {
(11)        ResultSetMetaData rsmd = rs.getMetaData();
(12)        int noColumns = rsmd.getColumnCount();
(13)        for (int i = 1; i < noColumns; i++) {
(14)            System.out.print(rsmd.getColumnLabel(i) + "\t");
(15)        }
(16)        System.out.println();
(17)
(18)        while (rs.isLast() == false) {
(19)            rs.next();
(20)            for (int i = 1; i < noColumns; i++) {
(21)                System.out.print( rs.getObject(i)+"\t" );
(22)            }
(23)            System.out.println();
(24)        }
(25)    }
(26)
(27)    public static void main(String[] args) {
(28)        try {
(29)            Class.forName("com.mysql.jdbc.Driver");
(30)        } catch (ClassNotFoundException e) {
(31)            System.err.println("Driver class not found");
(32)            e.printStackTrace();
(33)        }
(34)        Connection con = null;
(35)
(36)        try {
(37)            con =
(38)                (Connection) DriverManager.getConnection(
(39)                    "jdbc:mysql://localhost/jdbctest/",
(40)                    "mario",
(41)                    "thePassword");
(42)        } catch (SQLException e1) {
(43)            System.err.println("Error establishing database connection");
(44)            e1.printStackTrace();
(45)        }

```



```

(46)
(47)         Statement stmt = null;
(48)         try {
(49)             stmt = (Statement) con.createStatement(ResultSet.
TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
(50)         } catch (SQLException e2) {
(51)             System.err.println("Error creating SQL-Statement");
(52)             e2.printStackTrace();
(53)         }
(54)
(55)         try {
(56)             ResultSet uprs = (ResultSet) stmt.executeQuery("SELECT *
FROM EMPLOYEE;");
(57)             printResultSet(uprs);
(58)             uprs.moveToInsertRow();
(59)             uprs.updateString("FNAME", "Mario");
(60)             uprs.updateString("LNAME", "Jeckle");
(61)             uprs.updateInt("SSN", 11111111);
(62)             uprs.insertRow();
(63)             uprs = (ResultSet) stmt.executeQuery("SELECT * FROM
EMPLOYEE;");
(64)             printResultSet(uprs);
(65)         } catch (SQLException e3) {
(66)             System.err.println("Error selecting values from table
EMPLOYEE");
(67)             e3.printStackTrace();
(68)         }
(69)     }
(70)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Auf dieselbe Weise können auch Tupel einer Relation verändert werden. Hierzu stehen eine Reihe von updateXXX-Methoden zur Verfügung, wobei XXX für den Typ des zu aktualisierenden Attributs steht.

Nach durchgeführter Modifikation der hauptspeicherresidenten Werte werden diese durch [updateRow](#) in die Datenbank rückgeschrieben.

Beispiel 56 zeigt dies:

Beispiel 56: Modifizieren von Daten

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import java.sql.Statement;
(5)
(6)import com.mysql.jdbc.Connection;
(7)
(8)public class JDBCSelect3 {
(9)    public static void main(String[] args) {
(10)        try {
(11)            Class.forName("com.mysql.jdbc.Driver");
(12)        } catch (ClassNotFoundException e) {
(13)            System.err.println("Driver class not found");
(14)            e.printStackTrace();
(15)        }
(16)        Connection con = null;
(17)
(18)        try {
(19)            con =
(20)                (Connection) DriverManager.getConnection(
(21)                    "jdbc:mysql://localhost/jdbctest/",
(22)                    "mario",
(23)                    "thePassword");
(24)        } catch (SQLException e1) {
(25)            System.err.println("Error establishing database connection");
(26)            e1.printStackTrace();
(27)        }
(28)
(29)        Statement stmt = null;

```



```

(30)         try {
(31)             stmt =
(32)                 (Statement) con.createStatement(
(33)                     ResultSet.TYPE_SCROLL_SENSITIVE,
(34)                     ResultSet.CONCUR_UPDATABLE);
(35)         } catch (SQLException e2) {
(36)             System.err.println("Error creating SQL-Statement");
(37)             e2.printStackTrace();
(38)         }
(39)
(40)         try {
(41)             ResultSet uprs =
(42)                 (ResultSet) stmt.executeQuery("SELECT * FROM
EMPLOYEE;");
(43)             int namePos = uprs.findColumn("LNAME");
(44)
(45)             while (uprs.isLast() == false) {
(46)                 uprs.next();
(47)                 if (uprs.getString(namePos).compareTo("Wallace") ==
0) {
(48)                     uprs.updateString(namePos, "Doe");
(49)                     uprs.updateRow();
(50)                 }
(51)             }
(52)
(53)         } catch (SQLException e3) {
(54)             System.err.println("Error selecting values from table
EMPLOYEE");
(55)             e3.printStackTrace();
(56)         }
(57)     }
(58)}

```

[Download des Beispiels](#)

Analog vollzieht sich der Löschvorgang mittels [deleteRow](#):

Beispiel 57: Löschen von Daten

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import java.sql.Statement;
(5)
(6)import com.mysql.jdbc.Connection;
(7)
(8)public class JDBCSelect4 {
(9)    public static void main(String[] args) {
(10)        try {
(11)            Class.forName("com.mysql.jdbc.Driver");
(12)        } catch (ClassNotFoundException e) {
(13)            System.err.println("Driver class not found");
(14)            e.printStackTrace();
(15)        }
(16)        Connection con = null;
(17)
(18)        try {
(19)            con =
(20)                (Connection) DriverManager.getConnection(
(21)                    "jdbc:mysql://localhost/jdbctest/",
(22)                    "mario",
(23)                    "thePassword");
(24)        } catch (SQLException e1) {
(25)            System.err.println("Error establishing database connection");
(26)            e1.printStackTrace();
(27)        }
(28)
(29)        Statement stmt = null;
(30)        try {
(31)            stmt =
(32)                (Statement) con.createStatement(
(33)                    ResultSet.TYPE_SCROLL_SENSITIVE,

```



```

(34)                                ResultSet.CONCUR_UPDATABLE);
(35)        } catch (SQLException e2) {
(36)            System.err.println("Error creating SQL-Statement");
(37)            e2.printStackTrace();
(38)        }
(39)
(40)        try {
(41)            ResultSet uprs =
(42)                (ResultSet) stmt.executeQuery("SELECT * FROM
EMPLOYEE;");
(43)            int namePos = uprs.findColumn("LNAME");
(44)
(45)            while (uprs.isLast() == false) {
(46)                uprs.next();
(47)                if (uprs.getString(namePos).compareTo("Smith") == 0)
{
(48)                    uprs.deleteRow();
(49)                }
(50)            }
(51)
(52)        } catch (SQLException e3) {
(53)            System.err.println("Error selecting values from table
EMPLOYEE");
(54)            e3.printStackTrace();
(55)        }
(56)    }
(57)}

```

[Download des Beispiels](#)

Die bisher betrachteten Varianten extrahieren Daten aus der Datenbank im Stile einer Momentaufnahme (*snapshot*) zum Zeitpunkt der Anfrage. Die einmal angefragten Inhalte können sich jedoch noch zur Laufzeit der zugreifenden JDBC-Applikation datenbankseitig ändern, wenn sie durch eine andere Applikation neu geschrieben werden. Zur Gewährleistung der Konsistenz des extrahierten Snapshots mit den tatsächlichen Datenbankinhalten steht die Operation [rowUpdated](#) zur Verfügung. Sie ermittelt ob der im Hauptspeicher befindliche Wert mit dem aktuellen Datenbankinhalt übereinstimmt, d.h. ob der DB-Inhalt aktualisiert wurde.

Beispiel 58 zeigt ein Umsetzungsbeispiel.

Beispiel 58: Test auf geänderte Daten

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import com.mysql.jdbc.Connection;
(5)import com.mysql.jdbc.Statement;
(6)
(7)public class JDBCSelect7 {
(8)    public static void main(String[] args) {
(9)        try {
(10)            Class.forName("com.mysql.jdbc.Driver");
(11)        } catch (ClassNotFoundException cnfe) {
(12)            System.err.println("Driver class not found");
(13)            cnfe.printStackTrace();
(14)        }
(15)        Connection con = null;
(16)
(17)        try {
(18)            con =
(19)                (Connection) DriverManager.getConnection(
(20)                    "jdbc:mysql://localhost/jdbctest/",
(21)                    "mario",
(22)                    "thePassword");
(23)        } catch (SQLException sqle) {
(24)            System.err.println("Error establishing database connection");
(25)            sqle.printStackTrace();
(26)        }
(27)
(28)        Statement stmt = null;
(29)        try {
(30)            stmt =

```



```

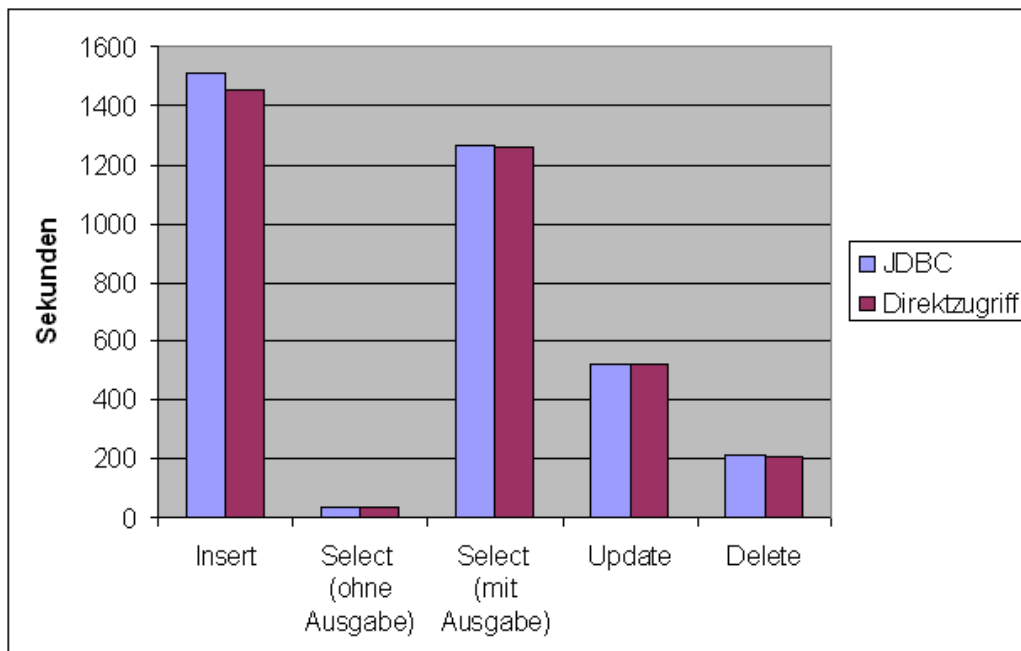
(31)                                (Statement) con.createStatement(
(32)                                ResultSet.TYPE_SCROLL_SENSITIVE,
(33)                                ResultSet.CONCUR_UPDATABLE);
(34)                                } catch (SQLException sqle) {
(35)                                    System.err.println("Error creating SQL-Statement");
(36)                                    sqle.printStackTrace();
(37)                                }
(38)
(39)                                try {
(40)                                    ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(41)                                    rs.absolute(5);
(42)                                    System.out.println(rs.getString("FNAME"));
(43)
(44)                                    System.out.println("sleeping ...");
(45)                                    Thread.sleep(6000);
(46)                                    System.out.println("awake ...");
(47)
(48)                                    if (rs.rowUpdated() == true) {
(49)                                        rs.refreshRow();
(50)                                        System.out.println(rs.getString("FNAME"));
(51)                                    }
(52)
(53)                                } catch (SQLException sqle) {
(54)                                    System.err.println("Error selecting values from table
EMPLOYEE");
(55)                                    sqle.printStackTrace();
(56)                                } catch (InterruptedException ie) {
(57)                                    ie.printStackTrace();
(58)                                }
(59)                                }
(60)}

```

[Download des Beispiels](#)

Performancebetrachtungen

Abbildung 5: JDBC-Geschwindigkeitsvergleich



(click on image to enlarge!)

Die Abbildung zeigt die Ergebnisse einiger Geschwindigkeitsmessungen als Vergleich zwischen dem Zugriff auf eine MySQL-Datenbank unter Nutzung der Textschnittstelle und der Abwicklung derselben Zugriffe mittels JDBC.

Zur Messung wurde eine nicht-indexierte Datenbank mit 10^7 Einträgen verwendet die aus der Relation tab bestand. Deren Tupel wurden aus Paaren von 36-Byte großen UUIDs gemäß dem [Spezifikationsentwurf der IETF](#) gebildet.

Zur Zeitmessung wurden folgende Einzeloperationen betrachtet:

- **Insert:** INSERT INTO tab VALUES(...)
Die Werte wurden unter Deaktivierung des Autocommit sequentiell eingefügt.
- **Select (ohne Ausgabe):** SELECT COUNT(*) FROM tab
Die ermittelte Gesamtzahl wurde nicht am Bildschirm ausgegeben.
- **Select (mit Ausgabe):** SELECT * FROM tab
Die Resultate wurden durch den Standardclient bzw. eine selbsterstellte (textmode) Java-Implementierung ausgegeben.
- **Update:** UPDATE tab SET UUID1="X" WHERE UUID1<>"X"
Durch die Initialisierung der Werte mit UUID-Einträgen wird sichergestellt, daß alle Tupel aktualisiert werden, da sie in keinem Fall den Wert X enthalten.
- **Delete:** DELETE FROM tab WHERE UUID2<>"X"
Durch die Initialisierung der Werte mit UUID-Einträgen wird sichergestellt, daß alle Tupel aktualisiert werden, da sie in keinem Fall den Wert X enthalten.

Insgesamt zeigt sich ein ausgewogenes Bild, in welchem der JDBC-Zugriff lediglich bei datenintensiven Zugriffen (große Mengen schreibender Zugriffe bei INSERT bzw. große Mengen lesender Operationen bei SELECT) im Bereich von fünf Prozent zurückliegt.

Diese enge Vergleichbarkeit der beiden Zugriffsmodi rührt von den Realisierung des eingesetzten JDBC-Treibers her; insbesondere von der Handhabung der physischen Datenbankverbindung auf Ebene des Netzwerkprotokolls.

SQL3-Datentypen

Die JDBC-API unterstützt mit Zugriffsmethoden auf die Datentypen BLOB, CLOB, ARRAY, Object und Ref bereits eine Untermenge des [SQL:1999-Standards](#). So können, vorausgesetzt das durch JDBC angesprochene DBMS unterstützt dies große unstrukturierte Binär- oder Textdaten sowie einfache verschachtelte Tabellen, mithin [NF²-Strukturen](#) verwaltet werden.

Beispiel 59 zeigt den Zugriff auf ein als eingebettete Tabelle realisiertes mengenwertiges Attribut. Die Beispieldatenbank wurde hierfür wie folgt modifiziert:

```
alter table EMPLOYEE ADD CAR SET('53M91','521R4', 'LL0415', 'XNU457');
update EMPLOYEE set CAR='XNU457' where SSN=123456789;
update EMPLOYEE set CAR='XNU457,521R4' where SSN="999887777";
```

Beispiel 59: Zugriff auf ein mengenwertiges Attribut

```
(1)import java.sql.Array;
(2)import java.sql.DriverManager;
(3)import java.sql.ResultSet;
(4)import java.sql.SQLException;
(5)import com.mysql.jdbc.Connection;
(6)import com.mysql.jdbc.Statement;
(7)
(8)public class JDBCSelect8 {
(9)    public static void main(String[] args) {
(10)        try {
(11)            Class.forName("com.mysql.jdbc.Driver");
(12)        } catch (ClassNotFoundException cnfe) {
(13)            System.err.println("Driver class not found");
(14)            cnfe.printStackTrace();
(15)        }
(16)        Connection con = null;
(17)
(18)        try {
(19)            con =
(20)                (Connection) DriverManager.getConnection(
(21)                    "jdbc:mysql://localhost/jdbctest/",
(22)                    "mario",
(23)                    "thePassword");
(24)        } catch (SQLException sqle) {
(25)            System.err.println("Error establishing database connection");
(26)            sqle.printStackTrace();
(27)        }
(28)
(29)        Statement stmt = null;
(30)        try {
(31)            stmt = (Statement) con.createStatement();
```



```

(32)    } catch (SQLException sqle) {
(33)        System.err.println("Error creating SQL-Statement");
(34)        sqle.printStackTrace();
(35)    }
(36)
(37)    try {
(38)        ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE;");
(39)        while (!rs.isLast()) {
(40)            rs.first();
(41)            System.out.print(rs.getString("FNAME") + "\t");
(42)            Array cars = rs.getArray("CAR");
(43)            ResultSet carsRS = cars.getResultSet();
(44)            System.out.print("(");
(45)            while (!carsRS.isLast()) {
(46)                rs.first();
(47)                System.out.print(carsRS.getString("CAR"));
(48)                carsRS.next();
(49)            }
(50)            System.out.println(")");
(51)            rs.next();
(52)        }
(53)    } catch (SQLException sqle) {
(54)        System.err.println("Error selecting values from table
EMPLOYEE");
(55)        sqle.printStackTrace();
(56)    }
(57) }
(58)}

```

[Download des Beispiels](#)

Das Beispiel unterstreicht die Rolle der mengenwertigen Attribute als eingebettete Tabellen. So erfolgt der Zugriff auf die Einzelwerte des Attributs CAR identisch zur Ermittlung der Resultatmenge der SQL-Anfrage mittels `getResultSet`. Auch die Traversierung der einzelnen CAR-Elemente erfolgt äquivalent.

Die Aufnahme der *large objects* in ihrer Ausprägungsform als *Character Large Objects* (CLOB) oder *Binary Large Objects* (BLOB) stellen eine der zentralen Erweiterungen des SQL:1999-Standards gegenüber seinen Vorgängern dar.

Zwar ist die Ablage großer unstrukturierter Datenobjekte in relationalen Datenbanken konzeptionell durchaus diskussionswert, jedoch in der Praxis oftmals, trotz der teilweise erheblichen Geschwindigkeitseinbußen im Zugriff (so benötigt die Ausführung der Beispielapplikation mit einem 10^6 Byte großen Datenstrom 1,1 Sekunden, während dieselbe Operation dateisystembasiert in 0,1 Sekunde abläuft), gewünscht.

Beispiel 60 zeigt die notwendigen Schritte zur Ablage und erneuten Auslese eines aus einer Datei gewonnen Binärdatenstroms in der Datenbank.

Die Beispieldatenbank wurde hierfür um ein Attribut zur Aufnahme binärer Daten erweitert:

```
ALTER TABLE EMPLOYEE ADD binData blob;
```

Beispiel 60: Verarbeitung unstrukturierter Binärdaten

```

(1)import java.io.File;
(2)import java.io.FileInputStream;
(3)import java.io.FileOutputStream;
(4)import java.io.IOException;
(5)import java.sql.DriverManager;
(6)import java.sql.PreparedStatement;
(7)import java.sql.ResultSet;
(8)import java.sql.SQLException;
(9)
(10)import com.mysql.jdbc.Connection;
(11)import com.mysql.jdbc.Statement;
(12)
(13)public class JDBCSelect9 {
(14)    public static void main(String[] args) {
(15)        try {
(16)            Class.forName("com.mysql.jdbc.Driver");
(17)        } catch (ClassNotFoundException cnfe) {
(18)            System.err.println("Driver class not found");
(19)            cnfe.printStackTrace();
(20)        }

```




```

(21)         Connection con = null;
(22)
(23)         try {
(24)             con =
(25)                 (Connection) DriverManager.getConnection(
(26)                     "jdbc:mysql://localhost/jdbctest/",
(27)                     "mario",
(28)                     "thePassword");
(29)         } catch (SQLException sqle) {
(30)             System.err.println("Error establishing database connection");
(31)             sqle.printStackTrace();
(32)         }
(33)
(34)         try {
(35)             File file = new File(args[0]);
(36)             FileInputStream fis = new FileInputStream(args[0]);
(37)             PreparedStatement pstmt =
(38)                 con.prepareStatement(
(39)                     "UPDATE EMPLOYEE SET binData =? WHERE
SSN=123456789");
(40)             pstmt.setBinaryStream(1, fis, (int) file.length());
(41)             pstmt.executeUpdate();
(42)             fis.close();
(43)
(44)             //read it back from the database
(45)             Statement stmt = (Statement) con.createStatement();
(46)             ResultSet rs =
(47)                 stmt.executeQuery(
(48)                     "SELECT binData FROM EMPLOYEE WHERE
SSN='123456789'");
(49)
(50)             FileOutputStream fos = new FileOutputStream(args[1]);
(51)             if (rs.next())
(52)                 fos.write(rs.getBytes(1));
(53)             fos.close();
(54)
(55)         } catch (SQLException sqle) {
(56)             System.err.println("Error selecting values from table
EMPLOYEE");
(57)             sqle.printStackTrace();
(58)         } catch (IOException ioe) {
(59)             ioe.printStackTrace();
(60)         }
(61)     }
(62)}

```

[Download des Beispiels](#)

Die Binärdaten können naturgemäß nicht direkt in die SQL-UPDATE-Anweisung eingebunden werden, sie werden daher einer mittels `prepareStatement` vorerzeugten Anweisung durch Aufruf der Methode `setBinaryStream` übergeben.

Transaktionssteuerung

Zur Steuerung des transaktionalen Verhaltens einer JDBC-Anfrage bietet die Klasse `Connection` verschiedene Methoden an:

- Abfrage der aktuellen Isolationsstufe: `getIsolationLevel`.
Hierbei werden fünf Stufen unterschieden:
 - `TRANSACTION_NONE`: Keinerlei Transaktionsunterstützung
 - `TRANSACTION_READ_UNCOMMITTED`: Auch nicht durch commit freigegebene Daten werden gelesen.
Es können daher *dirty reads*, Nicht-wiederholbare- und Phantomlesevorgänge auftreten.
 - `TRANSACTION_READ_COMMITTED`: Nur durch commit freigegebene Daten werden gelesen.
nichtwiederholbare- und Phantomlesevorgänge können jedoch auftreten.
 - `TRANSACTION_REPEATABLE_READ`: Innerhalb einer Transaktion können die verarbeiteten Daten nicht durch eine andere Transaktion verändert werden. Das Auftreten von *dirty reads* und nichtwiederholbaren Lesevorgängen ist daher ausgeschlossen, Phantomlesevorgänge sind jedoch weiterhin möglich.
 - `TRANSACTION_SERIALIZABLE`: Strikte Isolation aller Transaktionen, auf dieser Stufe sind auch

Phantomlesevorgänge ausgeschlossen.

Jede Stufe geht zwar mit einer gesteigerten Qualität der durch eine Transaktion verarbeiteten Daten einher, jedoch senkt gleichzeitig eine striktere Isolationsstufe die Anzahl der gleichzeitigen Zugriffe auf die Datenbank und damit die Gesamtsystemperformance.

- Setzen der Isolationsstufe: [setTransactionIsolation](#)
- (De-)Aktivierung der automatischen Freigabe: [setAutoCommit](#). Die Übergabe von true bewirkt die Aktivierung des Modus bei dem jede Einzelanweisung sofort persistent übernommen und für andere Transaktionen sichtbar wird.
Standardmäßig ist diese Option aktiviert. Ihr aktueller Zustand kann per [getAutoCommit](#) ermittelt werden.
- Freigabe von Änderungen: [commit](#).
- Rücknahme von Änderungen: [rollback](#).
- Rücknahme von Änderungen bis zu definiertem Sicherungspunkt: [rollback\(Savepoint s\)](#).
- Setzen eines Sicherungspunktes: [setSavepoint](#).

Beispiel 1 zeigt die Nutzung des Transaktionskonzepts.

Zunächst wird die aktuelle Isolationsstufe ermittelt und geprüft ob das angesprochene DBMS die höchste durch JDBC vorhergesehene Isolationsstufe unterstützt.

Nach Abschaltung der automatischen Änderungsübernahme (`setAutoCommit(false)`) werden zunächst zwei Tupel in die Tabelle EMPLOYEE eingefügt, die jedoch nur innerhalb der laufenden Transaktion sichtbar werden, für alle anderen Transaktionen innerhalb des DBMS bleiben die neuen Werte (zunächst) unsichtbar.

Eine angenommene Fehlersituation führt zum Rücksetzen der Transaktion durch (`rollback`).

Nach Abschluß des Programms wurden zwar die beiden ersten Werte lokal in die Datenbank übernommen, aber noch innerhalb der laufenden Transaktion wieder daraus entfernt, weshalb sie zu keinem Zeitpunkt für andere Datenbankbenutzer sichtbar waren.

Beispiel 61: Transaktionsverarbeitung

```
(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)
(5)import com.mysql.jdbc.Connection;
(6)import com.mysql.jdbc.Statement;
(7)
(8)public class JDBCTransact1 {
(9)    private static void printContent(Statement stmt) throws SQLException {
(10)        ResultSet rs =
(11)            stmt.executeQuery("SELECT FNAME,MINIT,LNAME FROM EMPLOYEE;");
(12)        while (!rs.isLast()) {
(13)            rs.next();
(14)            System.out.println(
(15)                rs.getString("LNAME")
(16)                + "\t"
(17)                + rs.getString("MINIT")
(18)                + "\t"
(19)                + rs.getString("LNAME"));
(20)        }
(21)    }
(22)    public static void main(String[] args) {
(23)        try {
(24)            Class.forName("com.mysql.jdbc.Driver");
(25)        } catch (ClassNotFoundException cnfe) {
(26)            System.err.println("Driver class not found");
(27)            cnfe.printStackTrace();
(28)        }
(29)        Connection con = null;
(30)
(31)        try {
(32)            con =
(33)                (Connection) DriverManager.getConnection(
(34)                    "jdbc:mysql://localhost/jdbctest/",
(35)                    "mario",
(36)                    "thePassword");
(37)        } catch (SQLException sqle) {
(38)            System.err.println("Error establishing database connection");
(39)            sqle.printStackTrace();
(40)        }
(41)
(42)        Statement stmt = null;
(43)        try {
```



```

(44)         stmt = (Statement) con.createStatement();
(45)     } catch (SQLException sqle) {
(46)         System.err.println("Error creating SQL-Statement");
(47)         sqle.printStackTrace();
(48)     }
(49)
(50)     try {
(51)         int transactionIsolation = con.getTransactionIsolation();
(52)         switch (transactionIsolation) {
(53)             case Connection.TRANSACTION_NONE :
(54)                 System.out.println("Transactions are not
supported");
(55)                 break;
(56)             case Connection.TRANSACTION_READ_UNCOMMITTED :
(57)                 System.out.println(
(58)                     "Dirty reads, non-repeatable reads
and phantom reads can occur");
(59)                 break;
(60)             case Connection.TRANSACTION_READ_COMMITTED :
(61)                 System.out.println(
(62)                     "Dirty reads are prevented; non-
repeatable reads and phantom reads can occur");
(63)                 break;
(64)             case Connection.TRANSACTION_REPEATABLE_READ :
(65)                 System.out.println(
(66)                     "Dirty reads and non-repeatable
reads are prevented; phantom reads can occur");
(67)                 break;
(68)             case Connection.TRANSACTION_SERIALIZABLE :
(69)                 System.out.println(
(70)                     "Dirty reads, non-repeatable reads
and phantom reads are prevented");
(71)                 break;
(72)             }
(73)             if (transactionIsolation < Connection.
TRANSACTION_SERIALIZABLE) {
(74)                 con.setTransactionIsolation(
(75)                     Connection.TRANSACTION_SERIALIZABLE);
(76)                 if (con.getTransactionIsolation()
(77)                     != Connection.TRANSACTION_SERIALIZABLE) {
(78)                     System.out.println(
(79)                         "cannot set Connection.
TRANSACTION_SERIALIZABLE");
(80)                 } else {
(81)                     System.out.println(
(82)                         "reached highest possible isolation
level");
(83)                 }
(84)             }
(85)
(86)             con.setAutoCommit(false);
(87)             stmt.executeUpdate(
(88)                 "INSERT INTO EMPLOYEE VALUES
('Hans', 'X', 'Hinterhuber', '11111111', NULL, NULL, NULL, NULL, NULL, NULL);");
(89)             stmt.executeUpdate(
(90)                 "INSERT INTO EMPLOYEE VALUES
('Franz', 'X', 'Obermüller', '22222222', NULL, NULL, NULL, NULL, NULL, NULL);");
(91)             printContent(stmt);
(92)             //suppose error happens here
(93)             Thread.sleep(5000);
(94)             boolean error = true;
(95)             if (error) {
(96)                 con.rollback();
(97)             } else {
(98)                 stmt.executeUpdate(
(99)                     "INSERT INTO EMPLOYEE VALUES
('Fritz', 'X', 'Meier', '33333333', NULL, NULL, NULL, NULL, NULL, NULL);");
(100)            }
(101)            printContent(stmt);
(102)        } catch (SQLException sqle) {
(103)            sqle.printStackTrace();
(104)        } catch (InterruptedException ie) {

```

```

(105)                ie.printStackTrace();
(106)            }
(107)    }
(108)}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Neben dem Zurücksetzen einer vollständigen Transaktion bietet die JDBC-API auch die Möglichkeit alle Schritte bis zu einem anwenderdefinierten aus der Datenbank zu entfernen.

Beispiel 62 zeigt dies unter Verwendung der Methode `setSavepoint` zur Definition eines Sicherungspunktes und `rollback(sp)` zum Zurücksetzen bis zu diesem Sicherungspunkt.

Beispiel 62: Transaktionsverarbeitung mit Sicherungspunkten

```

(1)import java.sql.DriverManager;
(2)import java.sql.ResultSet;
(3)import java.sql.SQLException;
(4)import java.sql.Savepoint;
(5)
(6)import com.mysql.jdbc.Connection;
(7)import com.mysql.jdbc.Statement;
(8)
(9)public class JDBCTransact2 {
(10)    private static void printContent(Statement stmt) throws SQLException {
(11)        ResultSet rs =
(12)            stmt.executeQuery("SELECT FNAME,MINIT,LNAME FROM EMPLOYEE;");
(13)        while (!rs.isLast()) {
(14)            rs.next();
(15)            System.out.println(
(16)                rs.getString("LNAME")
(17)                    + "\t"
(18)                    + rs.getString("MINIT")
(19)                    + "\t"
(20)                    + rs.getString("LNAME"));
(21)        }
(22)    }
(23)    public static void main(String[] args) {
(24)        try {
(25)            Class.forName("com.mysql.jdbc.Driver");
(26)        } catch (ClassNotFoundException cnfe) {
(27)            System.err.println("Driver class not found");
(28)            cnfe.printStackTrace();
(29)        }
(30)        Connection con = null;
(31)
(32)        try {
(33)            con =
(34)                (Connection) DriverManager.getConnection(
(35)                    "jdbc:mysql://localhost/jdbctest/",
(36)                    "mario",
(37)                    "thePassword");
(38)        } catch (SQLException sqle) {
(39)            System.err.println("Error establishing database connection");
(40)            sqle.printStackTrace();
(41)        }
(42)
(43)        Statement stmt = null;
(44)        try {
(45)            stmt = (Statement) con.createStatement();
(46)        } catch (SQLException sqle) {
(47)            System.err.println("Error creating SQL-Statement");
(48)            sqle.printStackTrace();
(49)        }
(50)
(51)        try {
(52)            int transactionIsolation = con.getTransactionIsolation();
(53)            switch (transactionIsolation) {
(54)                case Connection.TRANSACTION_NONE :
(55)                    System.out.println("Transactions are not
supported");
(56)                    break;

```



```

(57)                                case Connection.TRANSACTION_READ_UNCOMMITTED :
(58)                                System.out.println(
(59)                                "Dirty reads, non-repeatable reads
and phantom reads can occur");
(60)                                break;
(61)                                case Connection.TRANSACTION_READ_COMMITTED :
(62)                                System.out.println(
(63)                                "Dirty reads are prevented; non-
repeatable reads and phantom reads can occur");
(64)                                break;
(65)                                case Connection.TRANSACTION_REPEATABLE_READ :
(66)                                System.out.println(
(67)                                "Dirty reads and non-repeatable
reads are prevented; phantom reads can occur");
(68)                                break;
(69)                                case Connection.TRANSACTION_SERIALIZABLE :
(70)                                System.out.println(
(71)                                "Dirty reads, non-repeatable reads
and phantom reads are prevented");
(72)                                break;
(73)                                }
(74)                                if (transactionIsolation < Connection.
TRANSACTION_SERIALIZABLE) {
(75)                                con.setTransactionIsolation(
(76)                                Connection.TRANSACTION_SERIALIZABLE);
(77)                                if (con.getTransactionIsolation()
(78)                                != Connection.TRANSACTION_SERIALIZABLE) {
(79)                                System.out.println(
(80)                                "cannot set Connection.
TRANSACTION_SERIALIZABLE");
(81)                                } else {
(82)                                System.out.println(
(83)                                "reached highest possible isolation
level");
(84)                                }
(85)                                }
(86)
(87)                                con.setAutoCommit(false);
(88)                                stmt.executeUpdate(
(89)                                "INSERT INTO EMPLOYEE VALUES
('Hans', 'X', 'Hinterhuber', '11111111', NULL, NULL, NULL, NULL, NULL);");
(90)                                Savepoint sp = con.setSavepoint();
(91)                                stmt.executeUpdate(
(92)                                "INSERT INTO EMPLOYEE VALUES
('Franz', 'X', 'Obermüller', '22222222', NULL, NULL, NULL, NULL, NULL);");
(93)                                printContent(stmt);
(94)                                //suppose error happens here
(95)                                Thread.sleep(5000);
(96)                                boolean error = true;
(97)                                if (error) {
(98)                                con.rollback(sp);
(99)                                }
(100)                               stmt.executeUpdate(
(101)                               "INSERT INTO EMPLOYEE VALUES
('Fritz', 'X', 'Meier', '33333333', NULL, NULL, NULL, NULL, NULL);");
(102)                               printContent(stmt);
(103)                               con.commit();
(104)                               } catch (SQLException sqle) {
(105)                               sqle.printStackTrace();
(106)                               } catch (InterruptedException ie) {
(107)                               ie.printStackTrace();
(108)                               }
(109)                               }
(110)    }

```

[Download des Beispiels](#)

Netzwerkverkehr

Der Netzwerkmitschnitt zeigt den TCP-Kommunikationsverlauf der SQL-Anfrage `SELECT * FROM`

<http://www.jeckle.de/vorlesung/eBusinessEng/script.html> (103 of 151)09.01.2004 07:42:54

```

0420 b 03 Y E S 0e 00 00 1d \t h a v e _ i s a m 03 Y E S \r 00 00 1e
\t h a v e
0440 _ r a i d 02 N O 16 00 00 1f \f h a v e _ s y m l i n k \b D
I S A B L
0460 E D 10 00 00 \f h a v e _ o p e n s s l 02 N O 15 00 00 ! 10
h a v e _
0480 q u e r y _ c a c h e 03 Y E S 0b 00 00 " \t i n i t _ f i
l l e 00 ( 00
04a0 00 # 1f i n n o d b _ a d d i t i o n a l _ m e m _ p o
o l _ s i
04c0 z e 07 2 0 9 7 1 5 2 ! 00 00 $ 17 i n n o d b _ b u f f e
r _ p o o
04e0 l _ s i z e \b 1 6 7 7 7 2 1 6 - 00 00 % 15 i n n o d b _
d a t a _
0500 f i l e _ p a t h 16 i b d a t a 1 : 1 0 M : a u t o e
x t e n d
0520 16 00 00 & 14 i n n o d b _ d a t a _ h o m e _ d i r 00 19
00 00 ' 16 i
0540 n n o d b _ f i l e _ i o _ t h r e a d s 01 4 18 00 00
( 15 i n n o
0560 d b _ f o r c e _ r e c o v e r y 01 0 1c 00 00 ) 19 i n n
o d b _ t
0580 h r e a d _ c o n c u r r e n c y 01 8 ! 00 00 * 1e i n n
o d b _ f
05a0 l u s h _ l o g _ a t _ t r x _ c o m m i t 01 1 18 00 00
+ 14 i n n
05c0 o d b _ f a s t _ s h u t d o w n 02 0 N 15 00 00 , 13 i n
n o d b _
05e0 f l u s h _ m e t h o d 00 1c 00 00 - 18 i n n o d b _ l o
c k _ w a
0600 i t _ t i m e o u t 02 5 0 17 00 00 . 13 i n n o d b _ l o
g _ a r c
0620 h _ d i r 02 . / 17 00 00 / 12 i n n o d b _ l o g _ a r c
h i v e 03
0640 0 F F 1f 00 00 0 16 i n n o d b _ l o g _ b u f f e r _ s
i z e 07 8
0660 3 8 8 6 0 8 1d 00 00 1 14 i n n o d b _ l o g _ f i l e _
s i z e 07
0680 5 2 4 2 8 8 0 1c 00 00 2 19 i n n o d b _ l o g _ f i l e
s _ i n _
06a0 g r o u p 01 2 1d 00 00 3 19 i n n o d b _ l o g _ g r o u
p _ h o m
06c0 e _ d i r 02 . / 1d 00 00 4 1a i n n o d b _ m i r r o r e
d _ l o g
06e0 _ g r o u p s 01 1 1a 00 00 5 13 i n t e r a c t i v e _ t
i m e o u
0700 t 05 2 8 8 0 0 18 00 00 6 10 j o i n _ b u f f e r _ s i z
e 06 1 3 1
0720 0 7 2 19 00 00 7 0f k e y _ b u f f e r _ s i z e \b 1 6 7
7 7 2 1 6
0740 L 00 00 8 \b l a n g u a g e B / o p t / r a i d / m y s
q l - s t
0760 a n d a r d - 4 . 0 . 1 2 - p c - l i n u x - i 6 8
6 / s h a r
0780 e / m y s q l / e n g l i s h / 17 00 00 9 13 l a r g e _
f i l e s
07a0 _ s u p p o r t 02 0 N 10 00 00 : \f l o c a l _ i n f i l
e 02 0 N 15
07c0 00 00 ; 10 l o c k e d _ i n _ m e m o r y 03 0 F F \b 00 00
< 03 l o g
07e0 03 0 F F 0f 00 00 = \n l o g _ u p d a t e 03 0 F F 0b 00 00 >
07 l o g _
0800 b i n 02 0 N 16 00 00 ? 11 l o g _ s l a v e _ u p d a t e
s 03 0 F F
0820 15 00 00 @ 10 l o g _ s l o w _ q u e r i e s 03 0 F F 11 00
00 A \f l o
0840 g _ w a r n i n g s 03 0 F F 13 00 00 B 0f l o n g _ q u e
r y _ t i
0860 m e 02 1 0 19 00 00 C 14 l o w _ p r i o r i t y _ u p d a
t e s 03 0
0880 F F 1b 00 00 D 16 l o w e r _ c a s e _ t a b l e _ n a m
e s 03 0 F

```

```

08a0 F 1b 00 00 E 12 m a x _ a l l o w e d _ p a c k e t 07 1 0
4 7 5 5 2
08c0 ! 00 00 F 15 m a x _ b i n l o g _ c a c h e _ s i z e \n
4 2 9 4 9
08e0 6 7 2 9 5 1b 00 00 G 0f m a x _ b i n l o g _ s i z e \n 1
0 7 3 7 4
0900 1 8 2 4 14 00 00 H 0f m a x _ c o n n e c t i o n s 03 1 0
0 16 00 00 I
0920 12 m a x _ c o n n e c t _ e r r o r s 02 1 0 17 00 00 J 13
m a x _ d
0940 e l a y e d _ t h r e a d s 02 2 0 1d 00 00 K 13 m a x _ h
e a p _ t
0960 a b l e _ s i z e \b 1 6 7 7 7 2 1 6 19 00 00 L \r m a x _
j o i n _
0980 s i z e \n 4 2 9 4 9 6 7 2 9 5 15 00 00 M 0f m a x _ s o r
t _ l e n
09a0 g t h 04 1 0 2 4 17 00 00 N 14 m a x _ u s e r _ c o n n e
c t i o n
09c0 s 01 0 12 00 00 O 0e m a x _ t m p _ t a b l e s 02 3 2 00
00 P 14 m a
09e0 x _ w r i t e _ l o c k _ c o u n t \n 4 2 9 4 9 6 7 2
9 5 * 00 00
0a00 Q 1f m y i s a m _ m a x _ e x t r a _ s o r t _ f i l
e _ s i z
0a20 e \t 2 6 8 4 3 5 4 5 6 % 00 00 R 19 m y i s a m _ m a x _
s o r t _
0a40 f i l e _ s i z e \n 2 1 4 7 4 8 3 6 4 7 1b 00 00 S 16 m y
i s a m _
0a60 r e c o v e r _ o p t i o n s 03 0 F F 00 00 T 17 m y i
s a m _ s
0a80 o r t _ b u f f e r _ s i z e 07 8 3 8 8 6 0 8 17 00 00 U
11 n e t _
0aa0 b u f f e r _ l e n g t h 04 8 1 9 2 14 00 00 V 10 n e t _
r e a d _
0ac0 t i m e o u t 02 3 0 13 00 00 W 0f n e t _ r e t r y _ c o
u n t 02 1
0ae0 0 15 00 00 X 11 n e t _ w r i t e _ t i m e o u t 02 6 0 \b
00 00 Y 03 n
0b00 e w 03 0 F F 13 00 00 Z 10 o p e n _ f i l e s _ l i m i t
01 0 F 00 00
0b20 [ \b p i d _ f i l e < / o p t / r a i d / m y s q l -
s t a n d
0b40 a r d - 4 . 0 . 1 2 - p c - l i n u x - i 6 8 6 / d a
t a / l i
0b60 n u x . p i d 0b 00 00 \ \t l o g _ e r r o r 00 \n 00 00 ] 04
p o r t 04
0b80 3 3 0 6 14 00 00 ^ 10 p r o t o c o l _ v e r s i o n 02 1
0 18 00 00 _
0ba0 10 r e a d _ b u f f e r _ s i z e 06 1 3 1 0 7 2 1c 00 00
` 14 r e a
0bc0 d _ r n d _ b u f f e r _ s i z e 06 2 6 2 1 4 4 14 00 00
a 11 r p l
0be0 _ r e c o v e r y _ r a n k 01 0 1a 00 00 b 11 q u e r y _
c a c h e
0c00 _ l i m i t 07 1 0 4 8 5 7 6 13 00 00 c 10 q u e r y _ c a
c h e _ s
0c20 i z e 01 0 14 00 00 d 10 q u e r y _ c a c h e _ t y p e 02
0 N \f 00 00
0c40 e \t s e r v e r _ i d 01 1 17 00 00 f 11 s l a v e _ n e t
_ t i m e
0c60 o u t 04 3 6 0 0 19 00 00 g 15 s k i p _ e x t e r n a l _
l o c k i
0c80 n g 02 0 N 14 00 00 h 0f s k i p _ n e t w o r k i n g 03 0
F F 17 00 00
0ca0 i 12 s k i p _ s h o w _ d a t a b a s e 03 0 F F 13 00 00
j 10 s l o
0cc0 w _ l a u n c h _ t i m e 01 2 17 00 00 k 06 s o c k e t
0f / t m p /
0ce0 m y s q l . s o c k 18 00 00 l 10 s o r t _ b u f f e r _
s i z e 06
0d00 5 2 4 2 8 0 0b 00 00 m \b s q l _ m o d e 01 0 0f 00 00 n 0b t
a b l e _

```



```

0d20 c a c h e 02 6 4 12 00 00 o \n t a b l e _ t y p e 06 I N N
0 D B 14 00
0d40 00 p 11 t h r e a d _ c a c h e _ s i z e 01 0 14 00 00 q \f
t h r e a
0d60 d _ s t a c k 06 1 9 6 6 0 8 1d 00 00 r \f t x _ i s o l a
t i o n 0f
0d80 R E P E A T A B L E - R E A D 0e 00 00 s \b t i m e z o n
e 04 C E S
0da0 T 18 00 00 t 0e t m p _ t a b l e _ s i z e \b 3 3 5 5 4 4
3 2 \r 00 00
0dc0 u 06 t m p d i r 05 / t m p / 1c 00 00 v 07 v e r s i o n 13
4 . 0 . 1
0de0 2 - s t a n d a r d - l o g 13 00 00 w \f w a i t _ t i m
e o u t 05
0e00 2 8 8 0 0 01 00 00 x p 11 00 00 00 03 S E T a u t o c o m m
i t = 1 03
0e20 00 00 01 00 00 00 18 00 00 00 03 S E L E C T * F R O M E M
P L O Y E
0e40 E ; 01 00 00 01 \n 19 00 00 02 \b E M P L O Y E E 05 F N A M E 03
\n 00 00 01 ý
0e60 03 01 00 00 19 00 00 03 \b E M P L O Y E E 05 M I N I T 03 01 00 00
01 p 03 00 00
0e80 00 19 00 00 04 \b E M P L O Y E E 05 L N A M E 03 \n 00 00 01 ý 03
01 00 00 17 00
0ea0 00 05 \b E M P L O Y E E 03 S S N 03 \t 00 00 01 03 03 01 00 00 19 00
00 06 \b E M
0ec0 P L O Y E E 05 B D A T E 03 \n 00 00 01 \n 03 00 00 00 1b 00 00 07 \b
E M P L O
0ee0 Y E E 07 A D D R E S S 03 1e 00 00 01 ý 03 00 00 00 17 00 00 \b \b E
M P L O Y
0f00 E E 03 S E X 03 01 00 00 01 p 03 00 01 00 1a 00 00 \t \b E M P L O Y
E E 06 S A
0f20 L A R Y 03 07 00 00 01 05 03 00 02 1c 00 00 \n \b E M P L O Y E E
\b S U P E
0f40 R S S N 03 \t 00 00 01 03 03 00 00 00 17 00 00 0b \b E M P L O Y E E
03 D N O 03
0f60 01 00 00 01 03 03 00 00 00 01 00 00 \f p R 00 00 \r 04 J o h n 01 B 05 S
m i t h \t
0f80 1 2 3 4 5 6 7 8 9 \n 1 9 6 5 - 0 1 - 0 9 18 7 3 1 F o
n d r e n
0fa0 , H o u s t o n , T X 01 M \b 3 0 0 0 0 . 0 0 \t 3 3
3 4 4 5 5
0fc0 5 5 01 5 R 00 00 0e \b F r a n k l i n 01 T 04 W o n g \t 3 3
3 4 4 5 5
0fe0 5 5 \n 1 9 5 5 - 1 2 - 0 8 15 6 3 8 V o s s , H o u
s t o n ,
1000 T X 01 M \b 4 0 0 0 0 . 0 0 \t 8 8 8 6 6 5 5 5 5 01 5 T
00 00 0f 06 A
1020 l i c i a 01 J 06 Z e l a y a \t 9 9 9 8 8 7 7 7 7 \n 1 9
6 8 - 0 7
1040 - 1 9 17 3 3 2 1 C a s t l e , S p r i n g , T X
01 F \b 2 5
1060 0 0 0 . 0 0 \t 9 8 7 6 5 4 3 2 1 01 4 W 00 00 10 \b J e n n
i f e r 01
1080 S 07 W a l l a c e \t 9 8 7 6 5 4 3 2 1 \n 1 9 4 1 - 0 6
- 2 0 17 2
10a0 9 1 B e r r y , B e l l a i r e , T X 01 F \b 4 3
0 0 0 . 0
10c0 0 \t 8 8 8 6 6 5 5 5 5 01 4 V 00 00 11 06 R a m e s h 01 K 07
N a r a y
10e0 a n \t 6 6 6 8 8 4 4 4 4 \n 1 9 6 2 - 0 9 - 1 5 18 9 7
5 F i r e
1100 0 a k , H u m b l e , T X 01 M \b 3 8 0 0 0 . 0 0
\t 3 3 3 4
1120 4 5 5 5 5 01 5 S 00 00 12 05 J o y c e 01 A 07 E n g l i s h
\t 4 5 3 4
1140 5 3 4 5 3 \n 1 9 7 2 - 0 7 - 3 1 16 5 6 3 1 R i c
e , H o u s
1160 t o n , T X 01 F \b 2 5 0 0 0 . 0 0 \t 3 3 3 4 4 5 5 5
5 01 5 S 00
1180 00 13 05 A h m a d 01 V 06 J a b b a r \t 9 8 7 9 8 7 9 8 7
\n 1 9 6 9

```

```

11a0 - 0 3 - 2 9 17 9 8 0   D a l l a s ,   H o u s t o
n ,   T X 01 M
11c0 \b 2 5 0 0 0 . 0 0 \t 9 8 7 6 5 4 3 2 1 01 4 G 00 00 14 05 J
a m e s 01
11e0 E 04 B o r g \t 8 8 8 6 6 5 5 5 5 \n 1 9 3 7 - 1 1 - 1 0
16 4 5 0
1200 S t o n e ,   H o u s t o n ,   T X 01 M \b 5 5 0 0 0 .
0 0 û 01 1
1220 01 00 00 15 þ

```

Die Analyse des Datenverkehrs zeigt, daß im Falle der JDBC-basierten Kommunikation ein gegenüber der nativen Schnittstelle um 3529 Byte vergrößertes Datenaufkommen ausgetauscht wird. Diese zusätzliche Datenmenge fällt jedoch nur einmal zum Zeitpunkt des JDBC-Verbindungsaufbaus statisch an. ([Vgl. Mitschnitt der mehrfachen Ausführung einer SQL-Anfrage innerhalb einer bestehenden JDBC-Verbindung](#))

Zusätzlich offenbart Zeile 0x40 des Datenverkehrs die verschlüsselte Übermittlung des Paßwortes des Anwenders *mario*. Allerdings werden die per Anfrage ermittelten Nutzdaten (ab Zeile 0xe40) unverschlüsselt über die Netzwerkschnittstelle übertragen und stellen somit ein potentiellies Angriffsziel dar.

Abhilfe hierfür kann die Tunnelung des Datenverkehrs, beispielsweise mittels [SSH](#), durch eine sichere Verbindung bieten.

Zugriff mit JDBC nicht nicht-relationale Quellen

Zwar legt die JDBC-Schnittstelle inhärent eine relationale und damit tabellenorientierte Sicht der zu verarbeitenden Datenquelle zugrunde, jedoch ist die Nutzung der JDBC keinesweges auf jeden Anwendungsfall beschränkt.

Beispielsweise steht mit dem Produkt [Sunopsis XML Driver \(JDBC Edition\)](#) ein JDBC-Treiber für den dateibasierten Zugriff auf XML-Inhalte zur Verfügung.

Der Code des Beispiels 63 zeigt die Verwendung.

Als Beispiel dient folgende XML-Datei:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<Employees>
  <Employee>
    <FName>John</FName>
    <Minit>B</Minit>
    <LName>Smith</LName>
  </Employee>
  <Employee>
    <FName>Franklin</FName>
    <Minit>T</Minit>
    <LName>Wong</LName>
  </Employee>
  <Employee>
    <FName>Alicia</FName>
    <Minit>J</Minit>
    <LName>Zelaya</LName>
  </Employee>
  <Employee>
    <FName>Jennnifer</FName>
    <Minit>S</Minit>
    <LName>Wallace</LName>
  </Employee>
  <Employee>
    <FName>Ramesh</FName>
    <Minit>K</Minit>
    <LName>Narayan</LName>
  </Employee>
  <Employee>
    <FName>Joyce</FName>
    <Minit>A</Minit>
    <LName>English</LName>
  </Employee>
</Employees>

```

Beispiel 63: JDBC-artige Verarbeitung von XML-Dateien

```

(1)import java.sql.Connection;
(2)import java.sql.DriverManager;
(3)import java.sql.ResultSet;
(4)import java.sql.SQLException;
(5)import java.sql.Statement;
(6)
(7)public class JDBCXML {
(8)    public static void main(String[] args) {
(9)        try {
(10)            Class.forName("com.sunopsis.jdbc.driver.xml.SnpsXmlDriver");
(11)        } catch (ClassNotFoundException cnfe) {
(12)            System.err.println("Driver class not found");
(13)            cnfe.printStackTrace();
(14)        }
(15)        Connection con = null;
(16)        try {
(17)            con = (Connection) DriverManager.getConnection("jdbc:snps:
xml");
(18)        } catch (SQLException e1) {
(19)            System.err.println("Error establishing database connection");
(20)            e1.printStackTrace();
(21)        }
(22)        Statement stmt = null;
(23)        try {
(24)            stmt = con.createStatement();
(25)        } catch (SQLException sqle) {
(26)            sqle.printStackTrace();
(27)        }
(28)        try {
(29)            stmt.execute("load file \"employee.xml\" on schema EMP
readonly");
(30)            stmt.execute("set schema EMP");
(31)            ResultSet rs = con.getMetaData().getTables("", "EMP", "%",
null);
(32)
(33)            while (rs.next()) {
(34)                System.out.println("TABLE=" + rs.getString
("TABLE_NAME"));
(35)                System.out.print("(");
(36)                ResultSet colRS =
con.getMetaData().getColumns(
(37)                    "",
(38)                    "EMP",
(39)                    rs.getString("TABLE_NAME"),
(40)                    "%");
(41)                while (colRS.next()) {
(42)                    System.out.println(colRS.getString
("COLUMN_NAME") + ",");
(43)                }
(44)                System.out.println(")\n");
(45)            }
(46)        } catch (SQLException sqle) {
(47)            sqle.printStackTrace();
(48)        }
(49)
(50)        ResultSet rs = null;
(51)        try {
(52)            rs = (ResultSet) stmt.executeQuery("SELECT * FROM
EMPLOYEE;");
(53)            while (!rs.isLast()) {
(54)                rs.next();
(55)                System.out.println(rs.getString("FNAME"));
(56)            }
(57)        } catch (SQLException sqle) {
(58)            sqle.printStackTrace();
(59)        }
(60)
(61)        try {
(62)            stmt.executeUpdate("INSERT INTO EMPLOYEE (FNAME, MINIT,
LNAME) VALUES('James', 'E', 'Borg');");
(63)        } catch (SQLException sqle) {
(64)            sqle.printStackTrace();
(65)

```



```

(66)          }
(67)
(68)      }
(69)}}

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Web-Referenzen 8: Weiterführende Links



- [JDBC @ SUN](#)
 - [JDBC learning center @ SUN](#)
 - [JDBC Tutorial](#)
 - [JDBC FAQ @ JGuru.com](#)
 - [G. Reese: Database Programming with JDBC and Java. O'Reilly, 1997](#)
 - [Verhältnis von X/Open CLI und ODBC](#)
-

▲ 3 Anwendungslogik und serverinterne Kommunikation

3.1 Java Remote Method Invocation (RMI)

Grundidee

Der Java-spezifische Ansatz der *Remote Method Invocation* (RMI) zielt darauf einen lokationstransparenten Aufruf innerhalb von (verteilten) Java-Anwendungen zu ermöglichen. RMI stellt hierbei die Einzelobjekte einer Applikation in den Vordergrund und gestattet den Aufruf von Methoden in derselben Art und Weise unabhängig davon ob sich das Objekt auf der aufrufenden Maschine selbst befindet oder an einer entfernten Lokation zur Verfügung steht. Damit stellt sich RMI in die Tradition des objektorientierten Verteilungsmechanismus CORBA, ohne allerdings dessen programmiersprachenunabhängige Zielsetzung zu verfolgen. Gleichzeitig wurde das von DCE übernommene Serialisierungsformat für Hauptspeicherresidente Inhalte auf die Objektübertragung mittels eines Netzprotokolls ausgedehnt.

Die Grundprimitive des Ansatzes ist daher der entfernte Methodenaufruf (engl. *Remote Procedure Call*, RPC) welcher Kraft der lokalisationstransparenten Abstraktion RMIs dem Programmierer dieselbe Aufrufschnittstelle für entfernte Methoden bietet, die bereits für die Interaktion mit lokalen Objekten zum Einsatz kommt.

Zusätzlich erweitert RMI bestimmte zunächst lokal angebotene Dienstleistungen, wie etwa automatische Speicherverwaltung (*Garbage Collection*) für die verteilte Anwendung mit der Zielsetzung die Verwendung entfernter Objekte der lokal vorliegender gleichzustellen (Lokalisationstransparenz).

Zusammenfassend formuliert die RMI-Spezifikation daher folgende Ziele:

- Transparenter Methodenaufruf von Objekten die durch verschiedene virtuelle Maschinen verwaltet werden.
- Integration des Verteilungsmodells in die Programmiersprache unter weitestgehendem Erhalt der Semantik.
- Transparente Anlage des verteilten Objektmodells.
- Schaffung eines einfachstmöglichen Mechanismus zur Erstellung verteilter Applikationen.
- Erhalt der durch die Laufzeitumgebung geschaffenen Typsicherheit.
- Unterstützung verschiedener Referenzierungssemantiken.
- Erhalt der Java-Sicherheitsumgebung auch für verteilt ablaufende Applikationen.
- Unterstützung einer Call-back-Schnittstelle für Applets.

Technische Grundkonzepte

Integrale Bestandteile des RMI-Ansatzes und direkte Folgerungen der angestrebten Transparenz sind Architekturbestandteile *Stub* und *Skeleton*. Beide erfüllen symmetrische Aufgabenstellungen und werden mittels eines Dienstprogrammes automatisch aus dem übersetzten Bytecode des für den entfernten Zugriff anzubietenden Objekts erstellt.

Zur Ausführungszeit wirkt der Stub als Stellvertreter des entfernten Serverobjektes innerhalb der Adreßumgebung des Clients, analog fungiert der Skeleton als serverseitiger Stellvertreter des aufrufenden Objekts.

Abbildung 21 zeigt den Ablauf des Aufrufs eines entfernten Objekts mittels RMI. Hierbei spielen die

Stub- und die Skeletonkomponente wie folgt zusammen:

1. Stub: Initiierung einer Verbindung mit derjenigen entfernt ablaufenden virtuellen Maschine welche das anzusprechende Objekt verwaltet.
2. Stub: Erzeugung der benötigten Leitungsdarstellung und Übertragung der Aufrufparameter an die entfernte virtuelle Maschine.
3. Stub: Warten auf Ausführungsende und evtl. erzeugten Rückgabeparameter.
4. Skeleton: Empfang und Decodierung der Leitungsdarstellung der Aufrufparameter.
5. Skeleton: Aufruf des (lokalen) Objekts und Entgegennahme etwaig erzeugter Rückgabewerte.
6. Skeleton: Erzeugung der benötigten Leitungsdarstellung und Übertragung der Rückgabewerte an die entfernte virtuelle Maschine.
7. Stub: Empfang und Decodierung der Leitungsdarstellung des etwaig übergebenen Rückgabewertes.
8. Stub: Übergabe des Rückgabewertes (falls vorhanden) an (lokales) Aufruferobjekt.



Abbildung 21: Ablauf des Aufrufs eines entfernten Objekts mittels RMI (click on image to enlarge!)

Um das entfernte Objekt lokalisieren zu können ohne das dem Aufrufer der Server auf dem dieses verwaltet wird bekannt sein muß wird als zusätzliche Architekturkomponente ein Verzeichnisdienst verwendet. Innerhalb dieses Dienstes wird jedes Objekt, auf das der entfernte Zugriff angeboten werden soll, ein eindeutiger Name und die physische Objektlokation verwaltet.

Unter Berücksichtigung des Verzeichnisdienstes ergibt sich daher folgender Interaktionsablauf:

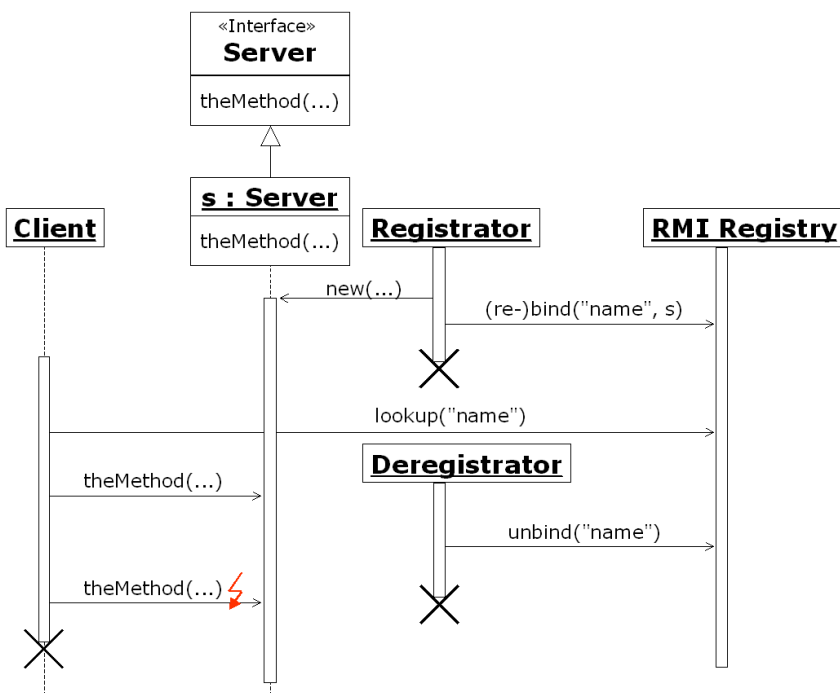


Abbildung 22: Operationen im Lebenszyklus eines per RMI zugreifbaren Objektes (click on image to enlarge!)

Das Bild zeigt das Objekt s als Instanz der Klasse Server sowie das auf die darin enthaltene Methode theMethod zugreifenden Client-Objekt. Zusätzlich ist mit RMI Registry der

Verzeichnisdienst zur Verwaltung der zugreifbaren Objekte dargestellt. Die beiden Objekte Registrator bzw. Deregistrator dienen zur Ablage (Bindung) des Server-Objektes an einen im Verzeichnisdienst gespeicherten Namen bzw. der Aufhebung dieser Bindung.

Da die Bindung objektbezogen erfolgt, muß die registrierende Instanz (im Beispiel Registrator) eine Referenz auf das abzulegende Objekt besitzen. Die Beispielinteraktion unterstellt daher die Erzeugung der benötigten Server-Instanz durch den Registrator selbst. Dieser bindet das Server-Objekt unter dem Klarnamen name und legt diese Zuordnung durch Aufruf der Methode [bind](#) oder [rebind](#) in der Registry ab.

Dort kann danach mittels der Methode [lookup](#), unter Angabe des Namens eine Referenz auf das gebundene Objekt, erfragt werden. Die Interaktion zeigt diesen Aufruf durch den Client.

Ausgehend von dieser Referenz kann durch den Client die durch die Server-Schnittstelle publizierte Methode theMethod aufgerufen werden.

Ferner zeigt der Ablauf die Deregistrierung des Server-Objektes durch das Deregistrator-Objekt mittels der Methode [unbind](#). Für diesen Aufruf ist keine Objektreferenz, sondern lediglich der Klarnamen des Objektes notwendig.

Vorgehen zur Entwicklung einer RMI-Applikation

Die Entwicklung einer RMI-basierten Applikation vollzieht sich immer im selben Ablauf der Erstellung der notwendigen Komponenten.

Dieser Ablauf ist nachfolgend anhand einer simplen Applikation veranschaulicht:

1. Definition der notwendigen Serverschnittstelle ([HelloInterface](#))
2. Implementierung des Servers ([HelloServer](#))
3. Implementierung des oder der Clientapplikationen ([HelloClient](#))
4. Erzeugung der Stub- und Skeletonklassen (durch das Werkzeug rmic)
5. Start des RMI-Verzeichnisdiensts (rmiregistry)
6. Registrierung des Serverobjektes
7. Start des Servers
8. Ausführung des Clients

Die Schnittstelle HelloInterface:

Beispiel 64: Die Schnittstelle HelloInterface



```
(1)import java.rmi.Remote;
(2)import java.rmi.RemoteException;
(3)
(4)public interface HelloInterface extends Remote {
(5)    public String sayHello() throws RemoteException;
(6)}
```

[Download des Beispiels](#)

Die Schnittstelle enthält die Operationen aller Methoden, die ein Serverobjekt zur Verwendung zur Verfügung stellen soll.

Hierbei sind naturgemäß keine als private oder protected deklarierten Operationen zugelassen, da auf sie kein Zugriff möglich wäre.

Für jede Operation ist die möglicherweise auftretende Erzeugung einer Ausnahme des Typs [RemoteException](#) vorzusehen. Dieser Typ kapselt durch Kommunikationsfehler verursachten Ausnahmen.

Zusätzlich ist die gesamte Schnittstelle als Spezialisierung der Standardschnittstelle [Remote](#) zu definieren. Diese Schnittstelle definiert selbst keine Operationen, sondern dient lediglich zur Kennzeichnung von Schnittstellen, die entfernt aufrufbare Methoden versammeln.

Die Klasse HelloServer:

Beispiel 65: Die Klasse HelloServer



```

(1)import java.net.MalformedURLException;
(2)import java.rmi.Naming;
(3)import java.rmi.RemoteException;
(4)import java.rmi.server.UnicastRemoteObject;
(5)
(6)public class HelloServer
(7)    extends UnicastRemoteObject
(8)    implements HelloInterface {
(9)
(10)    protected HelloServer() throws RemoteException {
(11)        super();
(12)        try {
(13)            Naming.rebind("rmi://localhost:1099/HelloService", this);
(14)        } catch (RemoteException re) {
(15)            re.printStackTrace();
(16)        } catch (MalformedURLException mue) {
(17)            mue.printStackTrace();
(18)        }
(19)    }
(20)
(21)    public static void main(String argv[]) throws RemoteException {
(22)        new HelloServer();
(23)    }
(24)
(25)    public String sayHello() throws RemoteException {
(26)        return (new String("Hello world!"));
(27)    }
(28)}

```

[Download des Beispiels](#)

Die Klasse enthält neben der Methode der durch die Schnittstelle exponierten Operation sayHello auch im Konstruktor die Registrierung innerhalb des RMI-Verzeichnisdienstes. Dieser ist jedoch, ebenso so wie alle anderen möglicherweise codierten Methoden, die nicht Bestandteil der explizierten Schnittstelle sind, nicht von einem entfernten Objekt aus zugreifbar. Formal implementiert die Klasse mit HelloInterface die öffentlich zur Verfügung gestellten Operationen. Zusätzlich ist muß sie als Spezialisierung der API-Klasse [UnicastRemoteObject](#) definiert sein um durch die virtuelle Maschine zur Laufzeit als Objekt mit entfernten Zugriffen behandelt zu werden. Dies zieht von der Behandlung des lokalen Falles abweichende Handhabungen der Parameterübergabe sowie der Freispeicherverwaltung und -rückgewinnung nach sich.

Die Implementierung der durch die Schnittstelle bekanntgegebenen und später durch entfernte Objekte nutzbaren Operation erfolgt identisch zur herkömmlichen Umsetzung um lokalen Falle. Einzige äußerlich erkennbare Änderung gegenüber ist die Deklaration der möglicherweise bei der Ausführung auftretenden [RemoteException](#). Diese wird jedoch üblicherweise nicht aktiv durch den anwenderdefinierten Code ausgelöst, sondern wird durch die Java-Standardbibliothek bzw. die -Laufzeitumgebung erzeugt. Die Deklaration dieser Ausnahme ist für Operationen deren Implementierung entfernt zugegriffen werden soll zwingend.

Zusätzlich nimmt die Applikation im Konstruktor die Registrierung des neu erzeugten Objektes unter dem frei gewählten Namen HelloService in der RMI-Registry vor. Hierbei wird die Methode rebind verwendet, um eine eventuell bereits existierende Objektbindung unter diesem Namen zu überschreiben. Auffallend ist die zur Bindung in der Registry verwendete URI rmi://localhost:1099/HelloService. Sie enthält nach dem identifizierenden URI-Schema (rmi) zwingend die Identifikation derjenigen Maschine, die das bereitzustellende Objekt verwaltet. Einschränkend sei hier jedoch angemerkt, daß RMI lediglich die Registrierung von Objekten vorsieht, die auf derselben physischen Maschine verwaltet werden wie der Registrierungsdienst selbst. Daher ist neben dem Pseudonamen localhost lediglich der Name der Maschine bzw. deren IP-Adresse oder die Pseudoadresse 127.0.0.1 zugelassen. Die optionale Portnummer benennt die logische TCP-Portadresse über welche Kommunikationen mit der RMI-Registry abgewickelt werden. Daran im Anschluß folgt der anwenderdefinierte Namen des abzulegenden Objekts.

Obwohl der nach erfolgreicher Ausführung der Methode rebind die Kontrolle an das neu erzeugte Objekt der Klasse HelloServer zurückübergeben wird terminiert das Programm nach Abarbeitung des Konstruktors (und damit Abschluß der Anweisungen in main) nicht, sondern wird durch die RMI-Registry wartend auf Aufrufe der Methode sayHello im Hauptspeicher gehalten.

Die Klasse HelloClient:

Beispiel 66: Die Klasse HelloClient

```

(1)import java.net.MalformedURLException;
(2)import java.rmi.Naming;
(3)import java.rmi.NotBoundException;
(4)import java.rmi.RemoteException;
(5)
(6)public class HelloClient {
(7)    public static void main(String[] args) {
(8)        HelloInterface obj = null;
(9)        try {
(10)            obj = (HelloInterface) Naming.lookup("rmi://53.16.71.55:1099/HelloService");
(11)        } catch (MalformedURLException mue) {
(12)            mue.printStackTrace();
(13)        } catch (RemoteException re) {
(14)            System.err.println("Lookup error");
(15)            re.printStackTrace();
(16)        } catch (NotBoundException nbe) {
(17)            System.out.println("Object not bound to registry");
(18)            nbe.printStackTrace();
(19)        }
(20)        System.out.println("retrieved object's oid=" + obj);
(21)        try {
(22)            System.out.println(obj.sayHello());
(23)        } catch (RemoteException re) {
(24)            System.err.println("Error while invoking sayHello");
(25)            re.printStackTrace();
(26)        }
(27)    }
(28)}

```



[Download des Beispiels](#)

Die Klasse des Beispiels 66 enthält den notwendigen Code um auf eine entfernt verwaltete und abgelegte Ausprägung der Schnittstelle `HelloInterface` zuzugreifen. Zunächst wird vermöge der Methode `lookup` eine Referenz eines `HelloInterface`-konformen Objekts aus dem RMI-Verzeichnisdienst erfragt. Hierzu dient die bereits zur Ablage verwendete URI auch zur Ermittlung des Objekts durch das nutzungswillige Clientobjekt. Naturgemäß müssen sich Client- und Serverobjekt nicht auf derselben physischen Maschine befinden, sondern müssen lediglich durch eine bestehende Netzwerkverbindung miteinander kommunizieren können. Nach Erhalt der Referenz kann das entfernte Objekt einem durch die lokale virtuelle Maschine verwaltetem gleichgestellt verwendet werden.

Da das Server-Objekt auch nach Abarbeitung seiner Methode resident im Speicher der entfernten virtuellen Maschine verbleibt ermittelt jeder Client bei jedem Aufruf der Methode `lookup` denselben Verweis auf dasselbe physische Objekt. Aus diesem Grunde ist bei der Programmierung darauf zu achten, daß sich nebenläufig stattfindende Zugriffe nicht gegenseitig behindern oder inkorrekte Ergebnisse liefern.

Die **Nutzung des Verzeichnisdienstes ist keine notwendige Voraussetzung** zur Ausführung entfernter Methodenaufrufe. Besitzt der Client bereits eine Referenz auf das entfernte Objekt (etwa aus einem Aufruf durch dieses welcher die Objektreferenz als Parameter enthielt), so kann der entfernte Methodenaufruf auch ohne vorherige Ermittlung der Referenz aus dem RMI-Verzeichnisdienst erfolgen.

Die **Parameterübergabe zwischen Client und entferntem Serverobjekt** geschieht anders als in der Java-üblichen Weise, d.h. für [primitivwerte](#) Parameter als Wert und Objektwertige als Referenz, durchgängig durch Wertübergabe.

Dies liegt darin begründet, daß der innerhalb der virtuellen Maschine verwandte Referenzierungsmechanismus auf Objekte als Inhalte des Heaps der virtuellen Maschine, nicht über Rechnergrenzen hinweg portabel gestaltet ist. Durch den notwendigen Aufwand der durch die Stub- und Skeletonkomponente zu codierenden bzw. zu decodierenden Daten steigt bei komplexen Objektstrukturen (z.B. Bäumen) sowohl die CPU-Last der Client- als auch der Servermaschine sowie die Netzwerklast.

Einzige Ausnahme von dieser Mimik bilden per RMI zugreifbare (entfernte) Objekte selbst. Sie werden nicht serialisiert und übertragen, sondern transparent durch ihre Stub-Repräsentation ersetzt, die an ihrer statt übertragen wird. Dies gilt insbesondere natürlich auch für Selbstreferenzen des Serverobjektes, die dieses als Rückgabewert liefert.

Naturgemäß ergeben sich auch für die **verteilte Freispeicherverwaltung** (distributed garbage

collection) veränderte Rahmenbedingungen, wenngleich gemäß dem RMI-Designziel größtmöglicher Transparenz versucht wurde eine ähnliche Handhabung wie bereits für den lokalen Fall zu realisieren.

Das in RMI gewählte Vorgehen orientiert sich an dem in der Programmiersprache [Modula-3](#) realisierten: Jede virtuelle Maschine führt eine Tabelle mit einem Boole'schen Eintrag für jedes aufgrund der publizierten Schnittstellendefinition potentiell entfernt zugreifbare Objekt. Jeder Eintrag ist vorgabegemäß mit dem Wert *clean* belegt. Wird eine Referenz auf das Objekt erfragt, so wird dieses als *dirty* markiert. Erfolgt innerhalb einer gewissen Zeit (*lease Zeit*) keine erneute Anfrage nach einer weiteren Referenz oder ein Zugriff auf das Objekt, so wird die Markierung auf *clean* zurückgesetzt und das Objekt so zur Freigabe markiert.

Die Zeitspanne bis zur automatisierten Markierung als nicht mehr benötigtes Objekt kann durch die Java-Property `java.rmi.dgc.leaseValue` kontrolliert werden. Vorgabegemäß ist sie auf zehn Minuten gesetzt.

Die Activation-API

Neben der u.U. speicherplatz- und laufzeitaufwendigen „Dauerinstanziierung“ eines entfernt zugreifbaren Objektes, wie sie das vorherige Beispiel einführt, existiert die Möglichkeit Objekte bei Bedarf dynamisch erzeugen zu lassen.

Das nachfolgende Beispiel zeigt dies. Der Ablauf in Erstellung und Ausführung ähnelt dem zuvor für einfache RMI-Applikationen gezeigten, jedoch unter Nutzung einer zusätzlichen Ausführungskomponente.

1. Definition der notwendigen Serverschnittstelle ([ActivatableHelloInterface](#))
2. Implementierung des Servers ([ActivatableHelloServer](#))
3. Implementierung einer Applikation zur Registrierung des Servers ([ActivatableSetup](#))
4. Implementierung des oder der Clientapplikationen ([ActivatableHelloClient](#))
5. Erzeugung der Stub- und Skeletonklassen (durch das Werkzeug `rmic`)
6. Start des RMI-Verzeichnisdiensts (`rmiregistry`)
7. Start des RMI-Daemons (`rmid`)
8. Registrierung des Serverobjektes
9. Ausführung des Clients

Im Vergleich zum [vorhergehenden Ablauf](#) wurde in diesem Beispiel die Registrierung des Servers in eine eigenständige Applikation ausgelagert, da ihr Ablauf im Konstruktor des Serverobjektes bei wiederholter Instanziierung durch die RMI-Umgebung redundant ausgeführt würde.

Ferner entfällt der explizite Start des Serverobjektes, da diese Aufgabe nunmehr automatisiert durch das Framework übernommen wird.

Die Komponente welcher die Kontrolle und Durchführung der Startvorgänge obliegt, der RMI-Daemon, muß dafür einmalig zusätzlich durch den Anwender gestartet werden.

Die Schnittstelle `ActivatableHelloInterface`:

Beispiel 67: Die Klasse `ActivatableHelloInterface`



```
(1)import java.rmi.Remote;
(2)import java.rmi.RemoteException;
(3)
(4)public interface ActivatableHelloInterface extends Remote {
(5)    public String sayHello() throws RemoteException;
(6)}
```

[Download des Beispiels](#)

Seitens der Schnittstellenbeschreibung der exponierten Operationen ergeben sich durch den Übergang auf die servergesteuerte Instanziierung keine Änderungen.

Die Klasse `ActivatableHelloServer`:

Beispiel 68: Die Klasse `ActivatableHelloServer`



```
(1)import java.rmi.MarshalledObject;
(2)import java.rmi.RemoteException;
(3)import java.rmi.activation.Activatable;
(4)import java.rmi.activation.ActivationID;
(5)
(6)public class ActivatableHelloServer
(7)    extends Activatable
(8)    implements ActivatableHelloInterface {
(9)
(10)    public ActivatableHelloServer(ActivationID id, MarshalledObject data)
(11)        throws RemoteException {
(12)        super(id, 0);
(13)    }
(14)
(15)    public String sayHello() throws RemoteException {
(16)        return (new String("Hello world!"));
(17)    }
(18)}
```

[Download des Beispiels](#)

Seitens der Implementierung des für entfernte Aufrufe zur Verfügung gestellten Objekts ergeben sich durch die geänderte Instanziierungsmimik keine gravierenden Änderungen. Lediglich die geänderte Spezialisierung, statt UnicastRemoteObject wird jetzt von der Klasse [Remote](#) abgeleitet, fällt zunächst ins Auge. Diese Änderung ist notwendig, um dem Framework die Möglichkeit der Instanziierung zu eröffnen, die bei einer völlig freien Formulierung --- insbesondere des Konstruktors --- nicht möglich wäre. Im Kern definiert die Klasse Activateable, neben einer Reihe von Hilfsmethoden, vier verschiedene Konstruktoren zur Anlage von Objekten. Mindestens einer davon ist zwingend durch die abgeleitete Klasse zu implementieren um dem eine automatisierte Objekterzeugung zu erreichen. Hinsichtlich der Problemlogik, im Beispiel der Methode sayHello ergeben sich keine Änderungen.

Die Klasse ActivatableHelloClient:

Beispiel 69: Die Klasse ActivatableHelloClient

```
(1)import java.net.MalformedURLException;
(2)import java.rmi.Naming;
(3)import java.rmi.NotBoundException;
(4)import java.rmi.RemoteException;
(5)
(6)public class ActivatableHelloClient {
(7)    public static void main(String[] args) {
(8)        ActivatableHelloInterface obj = null;
(9)        try {
(10)            obj = (ActivatableHelloInterface) Naming.lookup
("rmi://53.16.71.55:1099/HelloObj");
(11)        } catch (MalformedURLException mue) {
(12)            mue.printStackTrace();
(13)        } catch (RemoteException re) {
(14)            System.err.println("Lookup error");
(15)            re.printStackTrace();
(16)        } catch (NotBoundException nbe) {
(17)            System.out.println("Object not bound to registry");
(18)            nbe.printStackTrace();
(19)        }
(20)        System.out.println("retrieved object's oid=" + obj);
(21)        try {
(22)            System.out.println(obj.sayHello());
(23)        } catch (RemoteException re) {
(24)            System.err.println("Error while invoking sayHello");
(25)            re.printStackTrace();
(26)        }
(27)    }
(28)}
```



[Download des Beispiels](#)

Wie bereits beim Server gezeigt, ergeben auch Clientseitig durch die Veränderte Erzeugungssemantik keine Änderungen in der Ablauflogik, abgesehen von durch die geänderten

Namen dieses Beispiels bedingte Modifikationen.

Die Klasse ActivatableSetup:

Beispiel 70: Die Klasse ActivatableSetup



```
(1)import java.net.MalformedURLException;
(2)import java.rmi.MarshalledObject;
(3)import java.rmi.Naming;
(4)import java.rmi.RMISecurityManager;
(5)import java.rmi.RemoteException;
(6)import java.rmi.activation.Activatable;
(7)import java.rmi.activation.ActivationDesc;
(8)import java.rmi.activation.ActivationException;
(9)import java.rmi.activation.ActivationGroup;
(10)import java.rmi.activation.ActivationGroupDesc;
(11)import java.rmi.activation.ActivationGroupID;
(12)import java.rmi.activation.UnknownGroupException;
(13)import java.util.Properties;
(14)
(15)public class ActivatableSetup {
(16)    public static void main(String argv[]) throws RemoteException {
(17)        System.setSecurityManager(new RMISecurityManager());
(18)        Properties props = new Properties();
(19)        props.put("java.security.policy", "policy");
(20)        ActivationGroupDesc.CommandEnvironment ace = null;
(21)        ActivationGroupDesc exampleGroup = new ActivationGroupDesc(props,
ace);
(22)        ActivationGroupID agi = null;
(23)        try {
(24)            agi = ActivationGroup.getSystem().registerGroup
(exampleGroup);
(25)        } catch (RemoteException re) {
(26)            re.printStackTrace();
(27)        } catch (ActivationException ae) {
(28)            ae.printStackTrace();
(29)        }
(30)        String location = "file:/home/mario/RMITest/activation/";
(31)        MarshalledObject data = null;
(32)        ActivationDesc desc =
(33)            new ActivationDesc(agi, "ActivatableHelloServer", location,
data);
(34)
(35)        ActivatableHelloInterface stub = null;
(36)        try {
(37)            stub = (ActivatableHelloInterface) Activatable.register
(desc);
(38)        } catch (UnknownGroupException uge) {
(39)            uge.printStackTrace();
(40)        } catch (RemoteException re) {
(41)            re.printStackTrace();
(42)        } catch (ActivationException ae) {
(43)            ae.printStackTrace();
(44)        }
(45)        try {
(46)            Naming.rebind("rmi://localhost:1099/HelloObj", stub);
(47)        } catch (RemoteException re) {
(48)            re.printStackTrace();
(49)        } catch (MalformedURLException mue) {
(50)            mue.printStackTrace();
(51)        }
(52)    }
(53) }
(54)}
```

[Download des Beispiels](#)

Die nachhaltigsten Veränderungen spiegelt die neu eingeführte Klasse zur Registrierung des Objekts für entfernten Zugriff wieder.

Sie parametrisiert und initialisiert zunächst den RMI-Daemon, der später im Bedarfsfalle die Erzeugung der benötigten Objekte übernimmt. Daran Anschließend wird ein konform zur

publizierten Schnittstelle umgesetztes Objekt gemäß dem bisherigen Verfahren innerhalb des RMI-Verzeichnisdienstes an einen frei gewählten Namen gebunden.

Der automatische Aktivierungsprozeß erfordert zur Laufzeit des RMI-Daemons den Zugriff auf ein lokales oder erreichbares Dateisystem. Für alle Fälle des Zugriffs auf ein nichtlokales Dateisystem muß daher eine geeignete Sicherheitseinstellung der Javaausführungsumgebung gewählt werden. Im Standardumfang der API bereits enthalten ist die Klasse [RMISecurityManager](#). Sie erlaubt es dynamisch geladenen Code innerhalb einer RMI-Umgebung auszuführen.

Zusätzlich kann über die Java-Property `java.security.policy` eine sog. *policy*-Datei referenziert werden, die nach Herkunft abgestufte Zugriffsrechte für signierte Inhalte definiert.

Für das wiedergegebene Beispiel wurde eine sehr lose Policy gewählt, die jeglichem Code den Zugriff auf alle Java-Funktionalitäten gestattet. ([Policy-File](#)) Mittels der Policyeigenschaften `com.sun.rmi.rmid.ExecPermission` und `com.sun.rmi.rmid.ExecOptionPermission` läßt sich eine noch feinere Zugriffssteuerung erreichen, wobei jedoch diese vorgabegemäß immer unabhängig von der Codeherkunft und einer evtl. existierenden Signatur zu definieren sind.

Die späteren Aktivierungsaufrufe werden innerhalb einer *Aktivierungsgruppe* gebündelt. Eine solche wird mittels einer Beschreibung (als Ausprägung der Klasse [ActivationGroupDesc](#)) hinsichtlich ihrer eindeutigen Identität und ihrer Initialisierungsdaten beschrieben (im vorliegenden Beispiel ist sie mit `null` initialisiert).

Diese Aktivierungsgruppe wird mit den durch die Policy-Datei definierten Rechten ausgestattet erzeugt.

Die notwendige eindeutige Identität wird durch das Aktivierungssystem mittels des Aufrufs der Methode [registerGroup](#) automatisch zur Verfügung gestellt. Hierbei gilt eine eindeutige Zuordnung zwischen Gruppenkennung (`ActivationGroupID`) und verwaltender virtueller Maschine. Soll daher ein Objekt oder eine ganze Gruppe in einer separaten virtuellen Maschine ablaufen, so genügt es, eine andere Gruppenkennung durch das System zu generieren und bei der nachfolgend diskutierten Registrierung zu übergeben.

Anschließend wird durch das Beschreibungsobjekt mit dem Klassennamen des automatisch zu instanzierenden Objektes (im Beispiel: `ActivatableHelloServer`) sowie der physischen Lokation der Klasse und etwaigen Initialisierungsdaten erzeugt und der Aktivierungsgruppe zugeordnet.

Als letzter Schritt erfolgt die Registrierung der durch die Aktivierungsbeschreibung beschriebenen Klasse von Objekten, um so ihre spätere Erzeugung zu gewährleisten. Dies geschieht durch Aufruf der statischen Methode [register](#) der als Parameter die erzeugte Aktivierungsbeschreibung übergeben wird.

Der Aufruf dieser Methode liefert den Stub des registrierten Objekts. Aufgrund der Stellvertreterrolle des Stubs (d.h. seiner Konformität zur Schnittstelle `Remote`) kann dieser zur Registrierung der Objekte in der RMI-Registry verwendet werden.

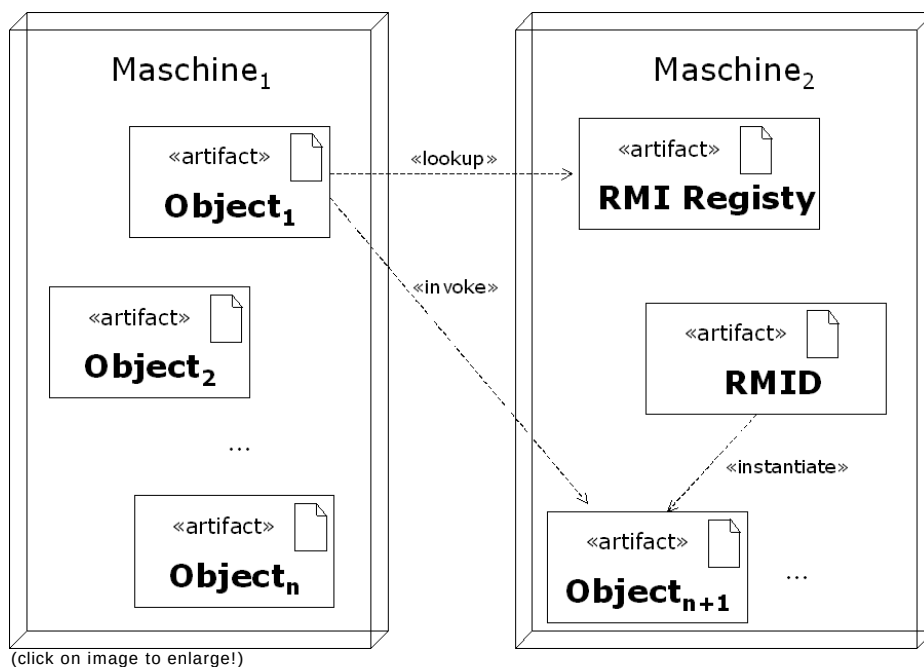
Der Aufruf der Klasse `ActivationSetup` erfolgt durch Übergabe von gültigen Belegungen der notwendigen Laufzeiteigenschaften an die virtuelle Javamaschine: `java -Djava.security.policy=./policy -Djava.rmi.codebase=file:/home/mario/RMITest/activation/ActivationSetup`.

Objekte werden zwar durch den RMI-Daemon automatisiert aktiviert, sobald ein Zugriff auf sie erfolgt (*lazy activation*), jedoch nicht mehr deaktiviert. Das bedeutet, daß sie bis zur Terminierung der virtuellen Maschine im Hauptspeicher verbleiben. Abhilfe schafft hier der explizite Aufruf der Methode [inactive](#) der ein bereits bestehendes Objekt passiviert, jedoch nicht aus der Registrierung entfernt.

Ähnlich wie bereits für die RMI-Registry angemerkt, muß auch der RMI-Daemon auf derselben physischen Maschine zur Ausführung gelangen, die das für entfernte Zugriffe bereitzustellende Objekt enthält.

Es ergibt sich daher generell folgendes Verteilungsschema:

Abbildung 6: Verteilungsstruktur einer RMI-Applikation



(click on image to enlarge!)

Um die irreführenden Begriffe *Client* und *Server* zu vermeiden sind die beiden Maschinen neutral durch Maschine₁ und Maschine₂ bezeichnet (Die Irreführung erklärt sich aus der mangelnden Trennschärfe der beiden Begriffe, sobald ein dienstanbieter Knoten auch gleichzeitig Dienste anderer Knoten nutzt, er damit gleichzeitig Server (seines Anfrages) und Client (bezüglich seiner Anfrage) ist).

Ein Object₁, das auf ein entferntes Object_{n+1} zugreift (invoke) kann optional mittels der RMI Registry, welche auf demselben Rechner wie die virtuelle Maschine die das Object_{n+1} verwaltet ablaufen muß, die notwendige Objektreferenz ermitteln (lookup). Sofern das zuzugreifende Objekt (noch) nicht existiert kann es über den Aktivierungsmechanismus gestartet werden. Die notwendige Objekterzeugung (instantiate) nimmt hierbei der RMI-Daemon (RMID) vor.

Sicherheitsgesichtspunkte

Generell werden die durch Stubs und Skeletons verarbeiteten Objekte oder Primitivwerte lediglich serialisiert, aber keine weiteren Maßnahmen zur Gewährleistung der Vertraulichkeit der übertragenen Daten ergriffen.

Durch die Integration des RMI-Mechanismus in die Java-API besteht die Möglichkeit innerhalb der Programmiersprache Möglichkeiten zur Übertragungssicherung zu ergreifen.

Seitens RMIs ist es hierbei vorgesehen die bestehende (einfache) TCP-Socketverbindung gegen beliebige selbsterstellte auszutauschen. Die Erzeugung der notwendigen Sockets muß mittels der Schnittstellen RMIServerSocketFactory bzw. RMIClientSocket abgewickelt werden. Auf diesem Wege stellt bleibt die Typsicherheit der Laufzeitumgebung, auch bei abgeänderten Transportprotokollen gewahrt.

Das nachfolgende Beispiel zeigt die Umsetzung einer einfachen gesicherten Verbindung mittels XOR-Bearbeitung aller versandten Inhalte.

Aus Gründen der Übersichtlichkeit sind die inhaltlich unverändert aus dem ersten Beispiel dieses Kapitels übernommenen Dateien [Hello.java](#) sowie [HelloClient.java](#) nicht wiedergegeben.

Die Klasse HelloImpl:

Beispiel 71: Die Klasse HelloImpl

```
(1)import java.rmi.Naming;
(2)import java.rmi.RMISecurityManager;
(3)import java.rmi.RemoteException;
(4)import java.rmi.server.UnicastRemoteObject;
(5)
(6)public class HelloImpl extends UnicastRemoteObject implements Hello {
(7)    public HelloImpl(String protocol, byte[] pattern) throws RemoteException {
(8)        super(
(9)            0,
(10)           new XorClientSocketFactory(protocol, pattern),
(11)           new XorServerSocketFactory(protocol, pattern));
(12)    }
(13)
(14)    public String sayHello() throws RemoteException {
```



```

(15)         StringBuffer sb = new StringBuffer();
(16)         for (int i=0;i<100;i++)
(17)             sb.append("X");
(18)         return sb.toString();
(19)     }
(20)
(21)     public static void main(String args[]) {
(22)
(23)         System.setSecurityManager(new RMISecurityManager());
(24)         byte[] aPattern = {(byte) 1011 };
(25)         try {
(26)             HelloImpl obj = new HelloImpl("xor", aPattern);
(27)             Naming.rebind("rmi://53.16.71.55:1099/HelloServer", obj);
(28)             System.out.println("HelloServer bound in registry");
(29)         } catch (Exception e) {
(30)             System.out.println("HelloImpl err: " + e.getMessage());
(31)             e.printStackTrace();
(32)         }
(33)     }
(34)}

```

[Download des Beispiels](#)

Die Klasse zeigt als einzige offensichtliche Veränderung die Ergänzung des Konstruktors des entfernten zugreifbaren Objekts um die Erzeugung der alternativ verwendeten XOR-Sockets; alle weiteren Aufwände die veränderte Kommunikation abzuwickeln sind für den Programmierer transparent und werden durch das RMI-Framework sowie die Implementierung der XOR-Sockets erbracht.

Die Bestandteile der XOR-Socketimplementierung finden sich hier:

- [XorClientSocketFactory](#)
- [XorInputStream](#)
- [XorOutputStream](#)
- [XorServerSocket](#)
- [XorServerSocketFactory](#)
- [XorSocket](#)

Web-Referenzen 9: Weiterführende Links



- [RMI @ SUN](#)
- [RMI Tutorial @ SUN](#)
- [Ninja RMI eine freie RMI-Implementierung](#)
- [JDBC FAQ @ JGuru.com](#)
- [G. Reese: Database Programming with JDBC and Java. O'Reilly, 1997.](#)
- [Verhältnis von X/Open CLI und ODBC](#)

3.2 Servlets

Ergebnisse eines Anwendungssystems werden typischerweise zur Laufzeit auf der Basis von Anwendungslogik und in Abhängigkeit der Eingaben berechnet. Dies gilt auch für Web-basierte Anwendungssysteme. Jedoch stellt das HTML-basierte Web konzeptionell zunächst ein statisches Medium, welches vorgefertigte Hypertextseiten auf über das HTTP-Protokoll übermittelte Anforderung ausliefert dar.

Zur Versöhnung dieses Widerspruchs haben sich zwei Ansätze herausgebildet: Zum einen die vollständige Verlagerung der Berechnungslogik und Erbebnispräsentation in den anfragenden Client durch Übertragung eines vollständigen binären Programms. Zum anderen der Aufruf von serverseitig vorgehaltenen Applikationen, die nicht im Adresskontext des Webserver operieren. Bekanntester Vertreter der ersten Kategorie sind die *Java Applets*, welche eine als *Java Bytecode* bezeichnete portable Binärrepräsentation an den Client übertragen, die dieser durch eine interpretativ arbeitende *virtuelle Maschine* ausführt. Die in der Praxis bedeutsamste Realisierung zum Zugriff auf extern vorliegende Programme stellt das *Common Gateway Interface* (CGI) dar.

Jedoch ergeben sich bei beiden Lösungsalternativen spezifische Probleme. So wird zur vollständigen Verlagerung der Ausführungslogik auf den Client, wie sie für Applets propagiert wird, die Möglichkeit zur Ausführung fremden Codes erzwungen. Dies läßt sich zwar durch geeignete Sicherheitsmechanismen absichern, jedoch bleibt die Notwendigkeit den Code per Internet zu

beziehen und lokal auszuführen.

Der CGI-Ansatz krankt an der mangelnden Ausführungsgeschwindigkeit, welche durch die notwendigen Prozeßkontextwechsel drastisch sinkt. Gleichzeitig unterliegt die auf diesem Wege angebundene Kontrolllogik nicht der Verwaltungshoheit des Servers.

Java *Servlets* stellen konzeptionell eine konfluente Weiterentwicklung der beiden Ansätze dar, die es sich zum Ziel gesetzt hat die angeführten Nachteile zu mildern.

Hierbei handelt es sich primär um Server-seitig ausgeführten Java-Code, der über Internetprotokolle (zumeist HTTP) mit dem Benutzer interagiert. Technisch gesehen stellen Servlets Module eines Web-Servers dar, die dynamisch anstelle der Auslieferung statischer Inhalte (z.B. HTML-Seiten) ausgeführt werden. Zur Ausführung dient eine als *Servlet Container* bezeichnete Softwarekomponente, welche die durch die dort abgelegten Servlets bedienten Schnittstellen der Standard-API bedient.

Bis zur Freigabe der Java-2-Plattform war die Servlet-API Bestandteil des JDK. Mittlerweile sind neben der Servlet-Spezifikation von SUN auch einige Implementierungen in verbreiteten Web-Servern verfügbar. Daher wurde die gesamte Servlet-API aus dem javax-Paket entfernt, und steht nunmehr nur noch als Bestandteil diverser Web-Server-Produkte (u.a. J2EE-Servern) zur Verfügung.

Lebenszyklus eines Servlets:

1. Der Server lädt das Servlet
2. Clientanforderungen werden durch das Servlet bedient
3. Der Server entlädt das Servlet (evtl. erst während des Server-Shutdown-Vorganges)

Im Detail korrespondieren diese drei Phasen mit Methoden der Servlet-Schnittstelle:

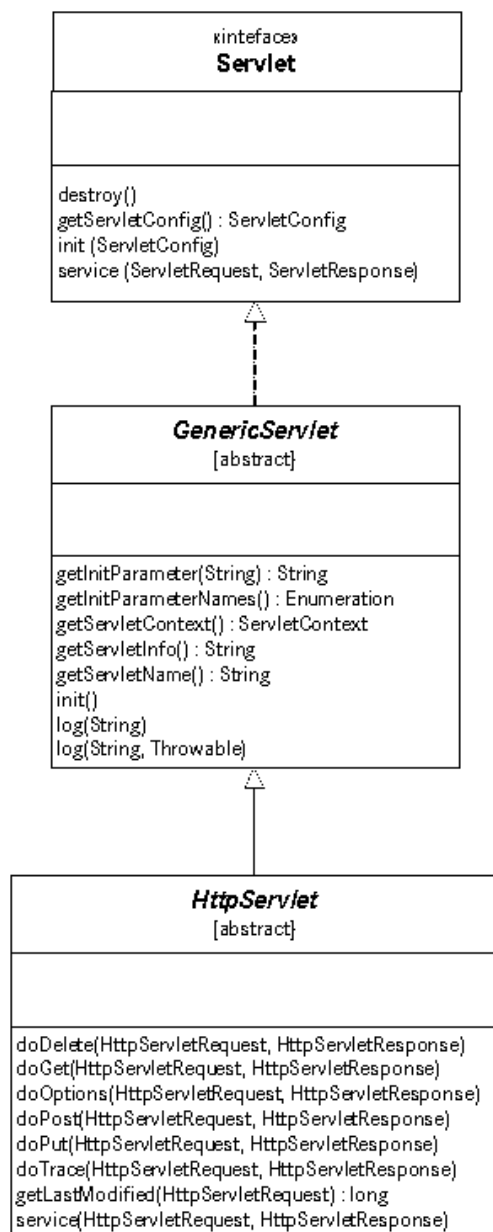
- `init(ServletConfig)`
Initialisierung des Servlets nach dem Ladevorgang, und vor dessen Ausführung.
- `service(ServletRequest, ServletResponse)`
Aufruf durch Servlet-Container (üblicherweise Web-Server) um Client-Anfrage zu bedienen. Pro Anfrage wird durch den Container vorgabegemäß ein neuer Thread gestartet. Beim Zugriff auf gemeinsam genutzte Ressourcen (Dateien, Datenbanken, etc.) ist daher zusätzlicher Synchronisationsaufwand notwendig.
- `destroy()`
Wird vom Servlet-Container bei der Deaktivierung aufgerufen. Zuvor haben alle noch laufenden Threads Gelegenheit ihre Ausführung normal zu beenden. Diese Methode sollte beanspruchte Ressourcen freigeben und ggf. Zustände persistent sichern. Nach Ablauf von `destroy()` erfolgen keine Aufrufe an `service()` mehr durch den Servlet Container.

Ergänzend existieren noch die beiden Methoden `getServletConfig()` zum Zugriff auf die bei der Initialisierung übergebenen Parameter, und `getServletInfo()` zur Rückgabe einer beliebigen Zeichenkette (zumeist für Copyrightinformation o.ä. benutzt).

Die Servlet API gliedert sich in drei Stufen. Die Schnittstelle `Servlet` definiert alle notwendigen Verwaltungsoperationen, die innerhalb eines Servlets durch Methoden zu implementieren sind. Gleichzeitig liefert die diese Operationen umsetzende abstrakte Klasse `GenericServlet` eine erste einfache Basisimplementierung dieser Operationen. Die Spezifikation empfiehlt konkrete protokollspezifische Servletimplementierungen immer als Spezialisierung dieser Klasse zu realisieren. Für das populäre HTTP-Protokoll liefern die Standardimplementierungen mit `HttpServlet` eine solche Implementierung mit.

Abbildung 1 stellt die Struktur der Basis API in der Übersicht zusammen.

Abbildung 7: Struktur der Servlet-API



Struktur eines Servlets:

Die Schnittstelle **Servlet** definiert alle Methoden, die ein Servlet zwingend implementieren muß. Dies kann durch eigenen Code geschehen, oder durch Erben von einer der vorgegebenen Klassen wie **GenericServlet** oder dem gebräuchlicheren **HttpServlet**.

Durch die Schnittstelle werden bereits Primitive für die beiden möglichen Kommunikationsrichtungen eingeführt: **ServletRequest** (Kommunikation vom Client zum Server) und **ServletResponse** (inverse Kommunikationsrichtung; Server an Client), als Parameter eines durch ein Servlet offerierten services.

Beide Primitiven sind wiederum als Schnittstellen definiert, die durch konkrete Implementierungen (wie **HttpServletRequest** bzw. **HttpServletResponse**) umgesetzt werden müssen.

Die Schnittstelle **ServletRequest** ermöglicht es dem Servlet auf Parameternamen, Protokolldetails und Rechnernamen der sendenden Maschine zuzugreifen.

Ferner stellt die Schnittstelle den **ServerInputStream** und den **PrintReader** zur Verfügung, über den Daten des Clients (beispielsweise per HTTP POST oder PUT) empfangen werden können.

Die Schnittstelle **ServletResponse** erlaubt es dem Servlet Verwaltungsinformation, wie MIME-Typ und Content-Länge, der Antwort zu setzen.

Darüberhinaus stellt es mit dem **ServletOutputStream** einen Kommunikationskanal für binäre Antwortdaten zur Verfügung. Unicode Text-artige Antworten können per **PrintWriter** übermittelt werden.

Die abstrakte **Klasse HttpServlet** stellt die notwendigen Methoden zur Bedienung von Client-Anforderungen bereit. Mindestens eine dieser Methoden ist zu überschreiben, um das gewünschte Verhalten des Servlets zu implementieren.

Die Implementierung des Beispiels 72 zeigt ein einfaches Servlet, welches nach einem Aufruf ein

HTML-Dokument erzeugt und zurückliefert.

Beispiel 72: Ein einfaches Servlet



```
(1)import java.io.IOException;
(2)import java.io.PrintWriter;
(3)import java.util.Date;
(4)import javax.servlet.ServletException;
(5)import javax.servlet.http.HttpServlet;
(6)import javax.servlet.http.HttpServletRequest;
(7)import javax.servlet.http.HttpServletResponse;
(8)
(9)public class HelloWorld extends HttpServlet {
(10)    public void doGet(HttpServletRequest request, HttpServletResponse response)
(11)        throws IOException, ServletException {
(12)        response.setContentType("text/html");
(13)        PrintWriter out = response.getWriter();
(14)        out.println("<html>");
(15)        out.println("<head>");
(16)        out.println("<title>Hello World!</title>");
(17)        out.println("</head>");
(18)        out.println("<body>");
(19)        out.println("<h1>Hello World!</h1>");
(20)        out.println("<p>Current Date: "+new Date().toString()+"</p>");
(21)        out.println("</body>");
(22)        out.println("</html>");
(23)    }
(24)}
```

[Download des Beispiels](#)

Im Beispiel wird die `doGet`-Methode überschrieben, d.h. das Servlet reagiert auf HTTP GET-Anfragen.

Als Antwort (ein wohlgeformtes HTML-Dokument) wird über den `PrintWriter` als Text (MIME-Typ `text/html`) versandt.

Der Servlet Container initialisiert und übergibt jedem Aufruf der Methode `doGet` ein Objekt des Typs `HttpServletRequest` und eines des Typs `HttpServletResponse` (welches im vorhergehenden Beispiel zur Übermittlung HTML-codierter Ergebnisse genutzt wurde).

Der Übergabeparameter des Typs `HttpServletRequest` ermöglicht den Zugriff auf die mit dem Aufruf des Servlets versandten Parameter. Beispiel 73 zeigt den Fall eines HTTP-basierten Servlets welches die Parameter der GET-Aufrufs eingebettet in die URL erhält. Ein möglicher Aufruf enthält daher neben der Lokation des Dienstes auch die notwendigen Aufrufparameter: `http://www.example.com/addService?a=1&b=2`.

Beispiel 73: Ein einfacher Dienst



```
(1)import java.io.IOException;
(2)import java.io.PrintWriter;
(3)
(4)import javax.servlet.http.HttpServlet;
(5)import javax.servlet.http.HttpServletRequest;
(6)import javax.servlet.http.HttpServletResponse;
(7)
(8)public class SimpleService extends HttpServlet {
(9)    public void doGet(HttpServletRequest req, HttpServletResponse resp)
(10)        throws IOException {
(11)        resp.setContentType("text/xml");
(12)        PrintWriter out = resp.getWriter();
(13)        out.println(
(14)            Integer.parseInt(
(15)                req.getParameter("a")
(16)                + Integer.parseInt(req.getParameter("b"))));
(17)    }
(18)}
```

[Download des Beispiels](#)

Neben den bisher genutzten GET-Anfragen stehen im Rahmen der Standard-API auch Methoden zur Behandlung der übrigen HTTP-Methoden zur Verfügung. Beispiel 74 zeigt die Funktionalität zum Zugriff auf Parameter im Rahmen eines DELETE-Aufrufs, auf HTTP-Headerdaten im Rahmen eines

GET-Aufrufs sowie die Extraktion von Nutzdaten eines POST-Aufrufs.

Beispiel 74: Verschiedene Servletmethoden

```
(1)import java.io.IOException;
(2)import java.io.PrintWriter;
(3)import java.util.HashMap;
(4)
(5)import javax.servlet.ServletInputStream;
(6)import javax.servlet.http.HttpServlet;
(7)import javax.servlet.http.HttpServletRequest;
(8)import javax.servlet.http.HttpServletResponse;
(9)
(10)public class Methods extends HttpServlet {
(11)    private HashMap hm;
(12)    public void init() {
(13)        hm = new HashMap();
(14)    }
(15)
(16)    public void doDelete(HttpServletRequest req, HttpServletResponse resp) {
(17)        hm.remove(req.getParameter("key"));
(18)    }
(19)
(20)    public void doGet(HttpServletRequest req, HttpServletResponse resp)
(21)        throws IOException {
(22)        resp.setContentType("text/html");
(23)        PrintWriter out = resp.getWriter();
(24)        out.println(hm.get(req.getHeader("key")));
(25)    }
(26)
(27)    public void doPost(HttpServletRequest req, HttpServletResponse resp) {
(28)        byte line[] = new byte[100];
(29)        String key = null;
(30)        String value;
(31)        ServletInputStream sip = null;
(32)        try {
(33)            sip = req.getInputStream();
(34)        } catch (IOException e) {
(35)            e.printStackTrace();
(36)        }
(37)        try {
(38)            sip.readLine(line, 0, 100);
(39)        } catch (IOException e1) {
(40)            e1.printStackTrace();
(41)        }
(42)        String lineStr = new String(line, 0, line.length);
(43)        key = lineStr.substring(lineStr.indexOf(":") + 1, lineStr.indexOf
(44)("&"));
(45)        value =
(46)            lineStr.substring(lineStr.lastIndexOf(":") + 1, lineStr.
(47)length());
(48)        hm.put(key, value);
(49)}
```



[Download des Beispiels](#)

Die Aufrufe erfolgen im HTTP-üblichen Stil:

- **GET**
GET /HashMapApp HTTP/1.0
key:name
- **POST**
POST /HashMapApp HTTP/1.0

key:name&value:mario
- **DELETE**
DELETE /HashMapApp?key=name

Prinzipiell ist der Servletgedanke auch auf beliebige andere Protokolle und deren Operationen übertragbar. Jedoch wird er in der Praxis zumeist für HTTP eingesetzt. Die API ist jedoch generell

so gestaltet, daß anwenderdefiniert jederzeit zusätzliche potokollspezifische Spezialisierungen der Klasse `GenericServlet` definiert werden können.

3.3 Enterprise Java Beans (EJB)

Neben den bereits aus anderen Veranstaltungen bekannten Servlets und den davon abgeleiteten Java Server Pages bildet die Technik der *Enterprise Java Beans* (EJB) einen weiteren zentralen Baustein der Java 2 Enterprise Plattform. Als serverseitige Komponenten kommt den EJBs heute große Bedeutung in der Realisierung komplexer Anwendungen, insbesondere durch Umsetzung der sog. „Business Logik“, d.h. den nicht-interaktiven fachlichen Anwendungsteilen, zu.

Der Begriff der Enterprise Java Bean stützt sich auf dem historisch älteren der *Java Bean*. Eine solche stellt eine abgeschlossene wiederverwendbare Softwarekomponente dar, die nach ihrer Erstellung über festgelegte Schnittstellen parametrisiert und manipuliert werden kann. Hierzu muß eine Bean eine festgelegte Interaktionsschnittstelle bieten, die durch die Java Bean Spezifikation definiert ist. Es handelt sich dabei um eine Reihe von Konventionen, der eine Bean gehorchen muß, jedoch um keine festgelegte API, die durch eine Komponente zu implementieren ist.

Der Begriff der Enterprise Java Bean greift diese inhaltliche Fundierung auf und präzisiert gleichzeitig die technische Umsetzung. So stellt eine Enterprise Java Bean eine Softwarekomponente dar, die in einer festgelegten Ausführungsumgebung, welche durch die EJB-Spezifikation festgelegte Dienste zur Nutzung durch die Beans anbieten kann. Eine solche Ausführungsumgebung wird als *Container* bezeichnet.

Ziel der Trennung in Komponente und Ausführungsumgebung ist die Zielsetzung die Enterprise Java Bean ausschließlich zur Umsetzung fachlicher Aufgaben heranzuziehen und alle infrastrukturellen Fragestellungen wie Betriebsmittelverwaltung, Persistenz oder Sicherheit durch die Ausführungsumgebung in gleicher Weise für alle Komponenten bereitzustellen.

Ein EJB-Container wird zumeist im Rahmen eines Application-Servers bereitgestellt.

Die gelegentlich anzutreffende Hervorhebung der anfänglich für Java Beans intendierten *visuellen Manipulationsmöglichkeit* trifft für Enterprise Java Beans nicht zu und hat sich für Java Beans auch nur begrenzte Bedeutung erlangt.

Spezifikationsgemäß können EJB-Container folgende Eigenschaften offerieren:

- **Betriebsmittelverwaltung**
Typischerweise verwaltet ein einziger EJB-Container gleichzeitig eine Reihe verschiedener Enterprise Java Beans. Zur Organisation und Aufrechterhaltung der Ausführbarkeit obliegt dem Container die Zuteilung von Betriebsmitteln wie Hauptspeicher, CPU-Zeit oder Netzwerkressourcen an die verwalteten EJB. Hierunter fällt insbesondere auch die Einlagerung, Instanziierung und Entfernung der EJBs selbst.
- **Zustandsverwaltung**
In praktischen Anwendungen ist oft die Nutzung zustandsbehafteter Kommunikation, die sich über verschiedene Einzelinteraktionen erstreckt gewünscht. Die hierfür notwendigen technischen Voraussetzungen (Zustandsspeicherung, Korrelation der Einzelinteraktionen) werden durch den Container bereitgestellt.
- **Transaktionsverwaltung**
Erweiterung der Zustandsverwaltung. Zur Gewährleistung des benötigten Verhaltens müssen EJBs keine eigenen Implementierungen zur Verfügung stellen, sondern können vorhandene Dienste des Containers nutzen.
- **Sicherheit**
Die Sicherheit von EJBs kann durch Vergabe von Zugriffsrechten und Rollen auf Containerebene gesteuert werden.
- **Persistenz**
Der interne Zustand einer verwalteten EJB kann wahlfrei in persistiert und zu einem späteren Zeitpunkt wiederhergestellt werden.
- **Entfernter Zugriff**
Der Zugriff auf EJBs erfolgt mittels [Remote Method Invocation](#) und ist daher Lokationstransparent.

Neben den in der Aufzählung dargestellten Eigenschaften dürfen Container zusätzlich Weitere wahlfrei implementieren.

EJB-Typen

Grundsätzlich lassen sich alle EJBs drei Typen zuordnen: *Session Beans*, *Entity Beans* und *Message Driven Beans*. Während erstere hauptsächlich zur Abbildung von Abläufen eingesetzt werden, dienen Entity Beans der Abwicklung von Zugriffen auf Daten. Eine Sonderstellung nehmen die *Message Driven Beans* ein, die lediglich hinsichtlich ihres Kommunikationsverhaltens festgelegt sind.

Session Beans dienen der Abbildung von Abläufen im Rahmen der Programmierung der sog. Business Logik. Die Lebensdauer (d.h. Zeitspanne zwischen Erzeugung im und Entfernung aus dem Hauptspeicher) ist daher identisch mit der einer durch den Client erfolgenden Anfrage. Jede zu einem Zeitpunkt existierende Session Bean repräsentiert daher eine zugehörige Clientinstanz. Nach ihrer internen Ausgestaltung werden *stateless* und *statefull* Session Beans unterschieden. Während Erstere keinen über einen einzigen Aufruf hinausgehenden Zustand verwalten und daher seiteneffektfrei lediglich auf den durch den Aufruf übermittelten Daten operieren erhält das zustandserhaltende Pendant die Daten eines Aufrufs und kann diese auch in nachfolgenden Aufrufen verarbeiten.

Entity Beans sind programmiersprachliche Stellvertreter datenbankresidenter Objekte. Sie dienen dem erleichterten Zugriff auf persistent vorliegende Datenbestände. Ihre interne Realisierung ist eng mit der Technik relationaler Datenbankmanagement System verbunden. So werden Sie durch einen anwenderdefinierten Primärschlüssel dauerhaft identifiziert.

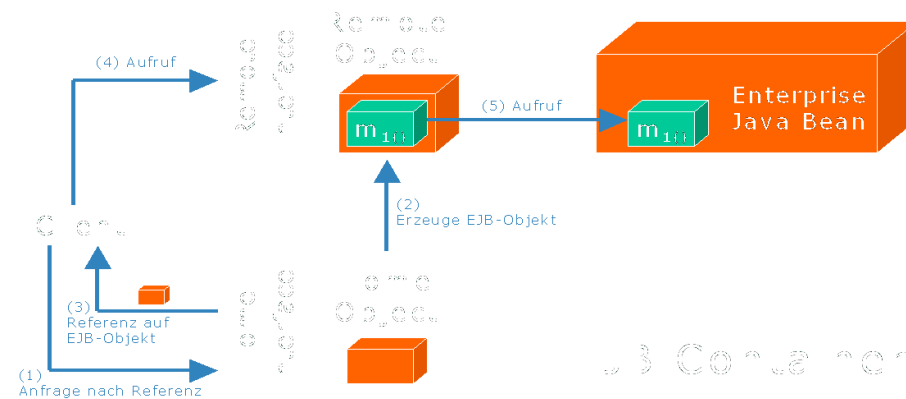
Message Driven Beans sind hinsichtlich ihres Kommunikationsverhaltens auf asynchrone Aufrufe beschränkt. Die Realisierung des eigentlichen Verhaltens wird durch eine Ausprägung eines der anderen Beantypen geboten.

Session Beans

Konzeptionell umfaßt jede EJB-Anwendung, die Session Beans einsetzt, die in [Abbildung 8](#) dargestellten Teile:

- Implementierung der EJB selbst.
- *Remote*-Schnittstelle zum Zugriff auf die durch die Bean publizierten Methoden.
- *Home*-Schnittstelle zur Ermittlung einer Referenz auf das Bean-Objekt.

Abbildung 8: Aufrufstruktur einer zustandslosen EJB



(click on image to enlarge!)

Gegenüber der Realisierung als RMI-Anwendung benötigt die Umsetzung als zustandslose Session Bean die Erstellung einer sog. „Home-Schnittstelle“ (*Home Interface*), welche die Operation *create* zur Instanziierung des serverseitigen EJB-Objekts bietet. Sie ist im Beispiel 75 dargestellt.

Beispiel 75: Home-Schnittstelle einer EJB

```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.CreateException;
(3)import javax.ejb.EJBHome;
(4)
(5)public interface SayHelloHome extends EJBHome {
(6)    public SayHello create() throws RemoteException, CreateException;
(7)}
```



[Download des Beispiels](#)

Die anwenderdefinierte Home-Schnittstelle erweitert die durch die Standard-API vorgegebene Schnittstelle *EJBHome*. Diese definiert Operationen zur Entfernung existierender EJB-Objekte aus dem Hauptspeicher (*remove*) sowie zur Ermittlung von Metadaten (*getEJBMetaData*) oder zum Erhalt eines netzwerkunabhängigen Verweises auf das EJB-Objekt (*getHomeHandle*).

Im Einzelnen sind dies die Operationen:

- *EJBMetaData* *getEJBMetaData()*
Liefert ein *EJBMetaData*-Objekt welches einzelne Eigenschaften einer EJB näher beschreibt. Hierzu zählen:

- Klasse der Home-Schnittstelle
- Klasse des Primärschlüssels (nur vorhanden sofern es sich um eine Entity Bean handelt)
- Klasse der Remote-Schnittstelle
- Boole'scher Wert, der angibt, ob es sich um eine Session Bean handelt
- Boole'scher Wert, der angibt, ob es sich um eine zustandslose Session Bean handelt
- `HomeHandle getHomeHandle()`
Liefert ein Objekt des Typs `HomeHandle` zurück, welches eine netzwerkunabhängige Abstraktion des Verweises auf das Home-Objekt realisiert.
- `void remove (Handle h)`
Entfernt ein durch den Objektverweis (`Handle`) identifiziertes EJB-Objekt aus dem Hauptspeicher.
- `void remove (Object pk)`
Entfernt ein durch das übergebene Primärschlüsselobjekt identifiziertes EJB-Objekt aus dem Hauptspeicher.

Interessanterweise definiert die Schnittstelle zwar Operationen zur Ermittlung von Daten über bestehende Objekte und zur Entfernung dieser Objekte aus dem Hauptspeicher, nicht jedoch zu ihrer Erzeugung.

Dies liegt in der durch die Programmiersprache Java angestrebten statischen Typsicherheit begründet, die es nicht gestattet Operationen mit variablen Parameterlisten --- wie sie für die zum API-Erstellungszeitpunkt unbekannten spezifischen Initialisierungsparameter aller denkbaren EJBs benötigt würden --- zu versehen.

Aus diesem Grunde definiert die EJB-Spezifikation informell, daß ein diese Schnittstelle erweiternde eigene Home-Schnittstelle zusätzlich die Methode `create`, deren Signatur als Rückgabotyp den Typ der Remote-Schnittstelle vorsehen muß definiert. Zusätzlich enthält diese Operation die zur Initialisierung der Bean benötigten Parameter in ihrer Parameterliste.

Die im Beispiel 76 ist Remote-Schnittstelle dargestellt, deren Ausprägungen von Home-Objekten angesprochen werden:

Beispiel 76: Remote-Schnittstelle einer EJB



```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.EJBObject;
(3)
(4)public interface SayHello extends EJBObject {
(5)    public String sayHello(String name) throws RemoteException;
(6)}
```

[Download des Beispiels](#)

Schnittstellen dieses Typs enthalten ausschließlich die fachlichen Operationen, d.h. die Signaturen der Methoden, die später durch den Client benutzt werden.

Jede Remote-Schnittstelle erweitert zusätzlich die vorgegebene Schnittstelle `EJBObject`, welche, ähnlich zur Home-Schnittstelle, einige Operationen zur Verwaltung eines EJB-Objektes vorgibt:

- `EJBHome getEJBHome()`
Liefert die Home-Schnittstelle einer EJB.
- `Handle getHandle()`
Liefert ein Objekt des Typs `HomeHandle` zurück, welches eine netzwerkunabhängige Abstraktion des Verweises auf das Home-Objekt realisiert.
- `Object getPrimaryKey()`
Liefert das Primärschlüsselobjekt einer Entity Bean.
- `boolean isIdentical(EJBObject eo)`
Prüft ob das übergebene EJB-Objekt dasselbe wie das Objekt ist auf dem die Methode ausgeführt wird.
- `void remove()`
Entfernt das EJB-Objekt aus dem Bean-Container.

Beispiel 77 zeigt die Implementierung der Bean selbst:

Beispiel 77: Realisierung einer Session Bean



```

(1)import java.rmi.RemoteException;
(2)import java.util.Date;
(3)import javax.ejb.EJBException;
(4)import javax.ejb.SessionBean;
(5)import javax.ejb.SessionContext;
(6)
(7)public class HelloWorldBean implements SessionBean {
(8)    public HelloWorldBean() {
(9)    }
(10)
(11)    public String sayHello(String name) {
(12)        return ("Hello " + name + " it is now " + new Date().toString());
(13)    }
(14)
(15)    public void setSessionContext(SessionContext arg0)
(16)        throws EJBException, RemoteException {
(17)    }
(18)    public void ejbRemove() throws EJBException, RemoteException {
(19)    }
(20)
(21)    public void ejbCreate() throws EJBException {
(22)    }
(23)
(24)    public void ejbActivate() throws EJBException, RemoteException {
(25)    }
(26)
(27)    public void ejbPassivate() throws EJBException, RemoteException {
(28)    }
(29)}

```

[Download des Beispiels](#)

Die programmiersprachliche Umsetzung der Bean enthält die Methoden der in der Remote-Schnittstelle bekanntgegebenen fachlichen Operationen. Zusätzlich muß ein Konstruktor expliziert werden, dessen Parameterliste mit den für die Operation create des Home-Interfaces gegebenen übereinstimmen.

Spezifikationsgemäß muß jede Session Bean die gleichnamige API-Schnittstelle implementieren. Diese definiert einige Operationen zur Behandlung unterschiedlicher Lebenszyklusstadien einer EJB. Hierunter fallen Methoden, die beim Erzeugen (ejbCreate), Entfernen (ejbRemove), bei der Passivierung (d.h. Auslagerung auf Hintergrundspeicher) (ejbPassivate) und dessen Reaktivierung (ejbActivate) eines EJB-Objekts durch die Ausführungsumgebung aufgerufen werden.

Bei der zwingend zu implementierenden Schnittstelle SessionBean handelt es sich nicht nur um eine Konvention um die Umsetzung der Lebenszyklusschnittstelle sicherzustellen, sondern auch um die Kategorisierung der Bean selbst. So stellt die im Beispiel verwandte Schnittstelle SessionBean neben EntityBean und MessageDrivenBean eine Spezialisierung der (operationslosen) Schnittstelle EnterpriseBean dar, deren „Implementierung“ durch eine Klasse lediglich zur Kennzeichnung dieser als EJB herangezogen wird.

Die genannten Spezialisierungen dieser Schnittstelle erfüllen daher sowohl den Zweck der Ausübung des Implementierungszwanges für die in ihnen aufgeführten Operationen als auch den der typisierenden Kennzeichnung.

Darüberhinaus ist EnterpriseBean als Spezialisierung der Standard-Schnittstelle [Serializable](#) angelegt. In der Konsequenz muß jedes EJB-Objekt durch die Javasprachmechanismen serialisierbar sein. Diese Eigenschaft wird insbesondere für die Passivierung und im Rahmen der Entity Beans genutzt.

Konzeptionell erinnert die Trennung in publizierte fachliche Schnittstelle (Remote-Schnittstelle) und deren technischer Umsetzung durch die EJB an die aus der Betrachtung des Remote Method Invocation Mechanismus bekannte Struktur.

Allerdings weicht die Umsetzung der Bean von der dort anzutreffenden Konvention ab die publizierte Schnittstelle selbst durch die realisierende Klasse zu implementieren. Dies liegt vor allem an der gegenüber RMI veränderten Struktur der publizierten Schnittstelle begründet. Während RMI für die Schnittstelle die Spezialisierung der operationslosen Standardschnittstelle [Remote](#) fordert die EJB-Spezifikation die Erweiterung der Schnittstelle [EJBObject](#), welche selbst die [oben dargestellten](#) Operationen definiert. Aus diesem Grunde würde die Aufnahme der Remote-Schnittstelle, obwohl konzeptionell durchaus zu rechtfertigen, in die Umsetzungsliste der EJB gleichzeitig die Implementierung von zumindest leeren Methodenrumpfen für die in EJBObject definierten Operationen notwendig werden lassen.

Abgesehen von dieser Ausnahme rekonstruiert das Verhältnis zwischen EJB und deren Remote-Schnittstelle die aus RMI bekannte Beziehung zwischen Schnittstelle und Umsetzung.

Die Nutzung einer durch eine Java Bean angebotenen Funktionalität erfolgt gemäß dem in [Abbildung 8](#) dargestellten Schema. Ein dies umsetzender Client ist in Beispiel 78 dargestellt.

Beispiel 78: Zugriff auf eine Session Bean



```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.CreateException;
(3)import javax.naming.Context;
(4)import javax.naming.InitialContext;
(5)import javax.naming.NamingException;
(6)import javax.rmi.PortableRemoteObject;
(7)
(8)public class CallHelloWorldBean {
(9)    public static void main(String[] args) {
(10)        try {
(11)            Context initial = new InitialContext();
(12)            Object objRef = initial.lookup("helloBean");
(13)
(14)            SayHelloHome home =
(15)                (SayHelloHome) PortableRemoteObject.narrow(
(16)                    objRef,
(17)                    SayHelloHome.class);
(18)            SayHello sh = home.create();
(19)
(20)            System.out.println(sh.sayHello("Mario"));
(21)
(22)        } catch (NamingException ne) {
(23)            ne.printStackTrace();
(24)        } catch (RemoteException re) {
(25)            re.printStackTrace();
(26)        } catch (CreateException ce) {
(27)            ce.printStackTrace();
(28)        }
(29)    }
(30)}
```

[Download des Beispiels](#)

Zunächst ermittelt der Client unter Nutzung der JNDI-API eine Referenz auf die EJB. Dies geschieht durch Anfrage (lookup) an den JNDI-Dienst unter Übergabe des bekannten Klarnamens (*helloBean*). Die erhaltene generische Referenz wird durch Aufruf der statischen Methode [narrow](#) der Klasse [PortableRemoteObject](#) typsicher in eine Ausprägung der Home-Schnittstelle konvertiert.

Der Aufruf der in dieser Schnittstelle durch den Anwender definierten create-Methode sorgt für die serverseitige Instanziierung der EJB, die als Ausprägung der Remote-Schnittstelle geliefert wird.

Tatsächlich wird nicht das EJB-Objekt selbst durch den Methodenaufruf retourniert, sondern lediglich ein netzwerktransparenter Verweis darauf, der jedoch clientseitig einer lokalen Objektreferenz gleichgestellt verwendet werden kann.

Ferner wird serverseitig zur Kommunikation mit der EJB ein Home-Objekt erzeugt, welchem eine Stellvertreterrolle für den anfragenden Client zukommt.

Der Aufruf der durch die EJB zur Verfügung gestellten Methode erfolgt identisch zu dem einer Lokalen.

Entity Beans

Die zweite zentrale Klasse von Enterprise Java Beans bilden die zur serverseitigen Persistierung von Objekten dienenden *Entity Beans*.

Sie kapseln Datenbankinhalte durch Objekte, die gemäß der EJB-Spezifikation instanzierbar und zugreifbar sind. Die Verwaltung der gekapselten Dateninhalte erfolgt durch ein frei festlegbares Datenbankmanagementsystem, die der Objekte selbst durch den EJB-Container.

Ziel dieser Technik ist es die Komplexität der Persistenzlogik für den Verwender der bereitgestellten Bean vollkommen transparent zu gestalten und serverseitig zu realisieren.

Die Familie der Entity Beans selbst zerfällt in zwei Untertypen, welche sich entlang des Realisierungspunktes der Persistenzlogik separieren: *Bean Managed Persistence*, der Bean obliegt die Umsetzung der Persistenzlogik, und *Container Managed Persistence*, hierbei wird die Persistenzlogik durch den EJB-Container realisiert.

Das nachfolgende Beispiel zeigt die Umsetzung einer Entity Bean mit Bean Managed Persistence. Es

kapselt die Verwaltung und den Zugriff auf Objekte, die Personen beschreiben. Jedes Personen-Objekt enthält Daten zu Name, Geburtsdatum und Wohnstraße. Der Name dient als eindeutige Identifikation und daher datenbankseitig als Primärschlüssel. Die notwendige Datenbanktabelle wurde erzeugt durch den SQL-Ausdruck:

```
CREATE TABLE PERSON( Name VARCHAR(20) PRIMARY KEY, Birthdate DATE, Street VARCHAR(30));
```

Wie bereits für die Realisierung von Session Beans eingeführt, werden auch zur Publikation der extern zugänglichen Schnittstellen Home und Remote Interfaces benötigt.

Struktur und Aufbau der Home-Schnittstelle ähnelt konzeptionell der für Session Beans eingeführten. Dieser Schnittstellentyp dient auch für Entity Beans zur Aufnahme der Verwaltungsoperationen zur Erzeugung (create) und zur Suche existierender EJBs (findByPrimaryKey).

Beispiel 79 zeigt die Home-Schnittstelle des Beispiels.

Beispiel 79: Home-Schnittstelle einer Entity Bean



```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.CreateException;
(3)import javax.ejb.EJBHome;
(4)import javax.ejb.FinderException;
(5)
(6)public interface PersonHome extends EJBHome {
(7)    public Person create(int year, int month, int day, String name, String
street) throws CreateException, RemoteException;
(8)    public Person findByPrimaryKey(String name) throws FinderException,
RemoteException;
(9)}
```

[Download des Beispiels](#)

Die Home-Schnittstelle zeigt die create-Operation zur Erzeugung einer neuen EJB-Instanz. Ihre Übergabeparameter dienen zur Konstruktion des neuen Objekts und werden durch die Bean-Implementierung interpretiert.

Ferner enthält die Schnittstelle mit findByPrimaryKey eine Operation deren Implementierung eine Entity-Bean anhand ihres Primärschlüssels identifiziert und liefert. Aus diesem Grunde erhält die Methode den zu suchenden Wert von Typ des Primärschlüssels übergeben.

Hinsichtlich der verwendeten Typen zeigt sich bereits hier, daß eine Abbildung der durch die Programmiersprache Java bereitgestellten Typen auf die des eingesetzten Persistenzsystems stattfinden muß. In der im Beispiel gewählten Ausführungsform der durch die EJB selbst verwalteten Persistenz muß diese Abbildung manuell durch den Programmierer bereitgestellt werden.

Die Remote-Schnittstelle gibt die für Nutzer der Bean zugänglichen Geschäftsfunktionen wieder. Daher enthält dieser Schnittstellentyp lediglich Operationen zum Zugriff auf die verwalteten Daten, nicht jedoch zur technischen Verwaltung und Interaktion mit dem Bean-Container.

Per Konvention muß diese Schnittstelle als Spezialisierung der Schnittstelle [EJBObject](#) definiert sein. Diese Standardschnittstelle definiert allgemeine Interaktionsformen, wie Löschen (remove), Vergleich (isIdentical) und Ermittlung des Primärschlüsselwertes (getPrimaryKey) die für alle Entity Bean Objekte gleichermaßen benötigt werden.

[isIdentical](#) liefert den Vergleich zweier serverseitiger EJB-Objekte und ermittelt so, ob zwei Java-Objektreferenzen auf dasselbe Datenbankobjekt zugreifen.

Mittels [getPrimaryKey](#) ermittelt den Wert des Primärschlüssels eines gegebenen EJB-Objekts aus der Datenbank.

Zur Lösung von datenbankresidenten Objekten wird [remove](#) eingesetzt. Der Aufruf dieser Methode entfernt ausschließlich die durch die EJB repräsentierten Datenbanktupel, die programmiersprachliche Objektrepräsentation bleibt jedoch über die gesamte Laufzeit (sofern nicht durch Gültigkeitsbereiche oder explizite NULL-Setzung explizit anders gehandhabt) intakt.

Beispiel 80: Remote-Schnittstelle einer Entity Bean




```

(1)import java.rmi.RemoteException;
(2)import javax.ejb.EJBObject;
(3)
(4)public interface Person extends EJBObject {
(5)    public int getAge() throws RemoteException;
(6)    public String getStreet() throws RemoteException;
(7)    public void setStreet(String street) throws RemoteException;
(8)}

```

[Download des Beispiels](#)

Beispiel 81 zeigt den vollständigen Code der Bean. Sie implementiert mit [EntityBean](#) die Standardschnittstelle alle Entity Beans, welche als Spezialisierung der ausschließlich markierenden Schnittstelle [EnterpriseBean](#) die notwendigen Basisoperationen zur Abwicklung der persistenten Speicherung.

Im Falle einer *Bean Managed Persistence* enthalten die Methoden der durch die Schnittstelle definierten und der zusätzlich im Rahmen der Spezifikation textuell definierten Operationen die notwendigen Aufrufe zur Ablage eines Objekts in der Datenbank und zu seiner späteren Extraktion daraus.

Im Einzelnen sind dies die Operationen:

Tabelle 24: Persistenzoperationen einer Entity Bean

Operation	Semantik	Zugehörige SQL-Anweisung
<code>ejbCreate</code>	Wird nach dem Erzeugen eines Java-Objektes aufgerufen um dieses in der Datenbank abzulegen. Diese Operation ist nicht Bestandteil der Schnittstelle, da ihre Parameter, die den Übergabeparametern des Objektkonstruktors entsprechen, zum Schnittstellenerzeugungszeitpunkt nicht feststehen.	INSERT
 <code>ejbFindByPrimaryKey</code>	Liefert den Wert des Primärschlüssels zurück, sofern ein Datenbankeintrag existiert, der durch diesen Primärschlüssel identifiziert wird. Diese Operation ist nicht Bestandteil der Schnittstelle, da ihre Parameter, die in Typ, Name und Reihenfolge der Zusammensetzung des Primärschlüssels entsprechen, zum Schnittstellenerzeugungszeitpunkt nicht feststehen. Diese Methode wird nicht durch den Anwender direkt aufgerufen, sondern stattdessen auf einer Ausprägung der Home-Schnittstelle das dort zur Verfügung stehende Analogon <code>findByPrimaryKey</code> , welches das durch den Primärschlüssel identifizierte EJB-Objekt zurückliefert. Diese Methode greift intern auf ausschließlich den Schlüssel liefernde Methode der Bean zu.	SELECT
ejbRemove	Entfernt das EJB-Objekt aus der Datenbank.	DELETE
ejbStore	Synchronisiert das Java-Objekt mit dem EJB-Objekt und aktualisiert so die Datenbankinhalte. Diese Methode wird nach jedem Zugriff mittels einer in der Remote-Schnittstelle aufgeführten Operation auf das EJB-Objekt ausgeführt.	UPDATE

Beispiel 81: Eine Entity Bean

```

(1)import java.rmi.RemoteException;
(2)import java.sql.Connection;
(3)import java.sql.DriverManager;
(4)import java.sql.ResultSet;
(5)import java.sql.SQLException;
(6)import java.sql.Statement;
(7)import java.util.Calendar;
(8)import java.util.GregorianCalendar;
(9)
(10)import javax.ejb.CreateException;
(11)import javax.ejb.EJBException;
(12)import javax.ejb.EntityBean;
(13)import javax.ejb.EntityContext;
(14)import javax.ejb.FinderException;
(15)import javax.ejb.RemoveException;
(16)
(17)public class PersonBean implements EntityBean {
(18)    //persistent fields
(19)    private String name;
(20)    private GregorianCalendar birthdate;
(21)    private String street;
(22)
(23)    public PersonBean() {
(24)    }
(25)
(26)    //methods which are part of the remote interface
(27)    public int getAge() {
(28)        return (
(29)            new GregorianCalendar().get(Calendar.YEAR)
(30)            - birthdate.get(Calendar.YEAR));
(31)    }
(32)    public String getStreet() {
(33)        return street;
(34)    }
(35)    public void setStreet(String street) throws RemoteException {
(36)        if (street.length() <= 30) {
(37)            this.street = street;
(38)        } else {
(39)            throw new RemoteException("cannot update street");
(40)        }
(41)    }
(42)    // -----
(43)    public String ejbCreate(
(44)        int year,
(45)        int month,
(46)        int day,
(47)        String name,
(48)        String street)
(49)        throws CreateException {
(50)        if (year > 1900
(51)            && month >= 1
(52)            && month <= 12
(53)            && day >= 1
(54)            && day <= 31
(55)            && name.length() <= 20
(56)            && street.length() <= 30) {
(57)
(58)            this.name = name;
(59)            this.birthdate = new GregorianCalendar(year, month, day);
(60)            this.street = street;
(61)            Statement stmt = getStatement();
(62)            String s =
(63)                new String(
(64)                    "INSERT INTO PERSON VALUES ('"
(65)                        + name
(66)                        + "', '"
(67)                        + birthdate.get(Calendar.YEAR)
(68)                        + "-"
(69)                        + birthdate.get(Calendar.MONTH)
(70)                        + "-"

```

```

(71)                + birthdate.get(Calendar.DATE)
(72)                + ",'"
(73)                + "'";
(74)                + street
(75)                + "'";
(76)                + ");");
(77)                try {
(78)                    stmt.executeUpdate(s);
(79)                } catch (SQLException e) {
(80)                    e.printStackTrace();
(81)                }
(82)            } else {
(83)                throw new CreateException("Invalid values supplied");
(84)            }
(85)            return name;
(86)        }
(87)    public void ejbPostCreate(
(88)        int year,
(89)        int month,
(90)        int day,
(91)        String name,
(92)        String street) {
(93)    }
(94)    // -----
(95)    public int ejbHomeGetAge() {
(96)        return 0;
(97)    }
(98)
(99)    public String ejbHomeGetStreet() {
(100)        return new String();
(101)    }
(102)
(103)    public void ejbHomeSetStreet(String street) {
(104)    }
(105)
(106)    public Statement getStatement() {
(107)        try {
(108)            Class.forName("com.mysql.jdbc.Driver");
(109)        } catch (ClassNotFoundException e) {
(110)            e.printStackTrace();
(111)        }
(112)        Connection con = null;
(113)        try {
(114)            con =
(115)                (Connection) DriverManager.getConnection(
(116)                    "jdbc:mysql://10.0.0.1/Address/",
(117)                    "mario",
(118)                    "thePassword");
(119)        } catch (SQLException e1) {
(120)            e1.printStackTrace();
(121)        }
(122)        try {
(123)            return ((Statement) con.createStatement());
(124)        } catch (SQLException e2) {
(125)            e2.printStackTrace();
(126)        }
(127)        return null; //never gets here
(128)    }
(129)
(130)    // -----
(131)    public void setEntityContext(EntityContext ectx)
(132)        throws EJBException, RemoteException {
(133)    }
(134)
(135)    public void unsetEntityContext() throws EJBException, RemoteException {
(136)    }
(137)
(138)    public void ejbRemove()
(139)        throws RemoveException, EJBException, RemoteException {
(140)        Statement stmt = getStatement();
(141)        String s = new String("DELETE FROM PERSON WHERE Name='" + name +
"','");

```



```

(142)         try {
(143)             stmt.executeUpdate(s);
(144)         } catch (SQLException e) {
(145)             e.printStackTrace();
(146)         }
(147)     }
(148)
(149)     public void ejbActivate() throws EJBException, RemoteException {
(150)     }
(151)
(152)     public void ejbPassivate() throws EJBException, RemoteException {
(153)     }
(154)
(155)     public void ejbLoad() throws EJBException, RemoteException {
(156)     }
(157)
(158)     public void ejbStore() throws EJBException, RemoteException {
(159)         Statement stmt = getStatement();
(160)         String s =
(161)             new String(
(162)                 "UPDATE PERSON SET Name='"
(163)                     + name
(164)                     + "', BDATE='"
(165)                     + birthdate.get(Calendar.YEAR)
(166)                     + "-"
(167)                     + birthdate.get(Calendar.MONTH)
(168)                     + "-"
(169)                     + birthdate.get(Calendar.DATE)
(170)                     + "', Street = '"
(171)                     + street
(172)                     + "' WHERE Name = '"
(173)                     + name
(174)                     + "';";
(175)         try {
(176)             stmt.executeUpdate(s);
(177)         } catch (SQLException e) {
(178)             e.printStackTrace();
(179)         }
(180)     }
(181)
(182)     public String ejbFindByPrimaryKey(String name) throws FinderException {
(183)         Statement stmt = getStatement();
(184)         String s =
(185)             new String("SELECT Name FROM PERSON WHERE Name='" + name +
(186)                 "';");
(187)         try {
(188)             ResultSet rs = stmt.executeQuery(s);
(189)             rs.first();
(190)             return rs.getString("Name");
(191)         } catch (SQLException e) {
(192)             e.printStackTrace();
(193)         }
(194)         return null; //never gets here
(195)     }

```

[Download des Beispiels](#)

Zusätzlich enthält die Bean des Beispiels mit `getAge` eine zwar in der Remote-Schnittstelle veröffentlichte Operation, die keinen direkt abgespeicherten Wert liefert, sondern diesen dynamisch zur Ausführungszeit anhand der verfügbaren Daten berechnet.

Alle anderen in der Remote-Schnittstelle aufgeführten Operationen (etwa: `getStreet`, `setStreet`) modifizieren lediglich, den durch die Attribute repräsentierten Java-Objektzustand und greifen nicht direkt auf die Datenbank zu.

Innerhalb der Datenbankzugreifenden Methoden muß durch den Anwender die Abbildung der Java-Datentypen auf die des verwendeten Datenbankmanagementsystems erfolgen. Die mit dem Präfix `ejb` versehenen Methoden zeigen dies für die lesenden und schreibenden DB-Zugriffe. So kann die im Beispiel für `name` und `street` verwendete Java-Repräsentation `String` vergleichsweise leicht in

den SQL-Typ VARCHAR abgebildet werden sofern durch alle Methoden, die Datenbankinhalte schreiben sicherstellen, daß nur zum Datenbankschema konforme Werte eingefügt werden. Die Beispielimplementierung zeigt dies exemplarisch anhand der Methoden `ejbCreate` und `setStreet`. Für programmiersprachliche Typen, die nicht direkt in DB-Typen abbildbar sind muß im Falle der Bean Managed Persistence der Bean-Entwickler selbst Sorge für die adäquate Abbildung tragen. Das Beispiel illustriert dies anhand des Java-Datumstyps [GregorianCalendar](#), der manuell in die durch das DBMS erwartete [ISO 8601](#)-konforme Darstellung zu überführen ist.

Einige der möglichen Interaktionen mit der Bean zeigt der Code des Clients aus Beispiel 82:

Beispiel 82: Client der auf eine Entity Bean zugreift

```
(1)import java.rmi.RemoteException;
(2)import javax.ejb.CreateException;
(3)import javax.ejb.FinderException;
(4)import javax.ejb.RemoveException;
(5)import javax.naming.Context;
(6)import javax.naming.InitialContext;
(7)import javax.naming.NamingException;
(8)import javax.rmi.PortableRemoteObject;
(9)
(10)public class CallPersonBean {
(11)    public static void main(String[] args) {
(12)        try {
(13)            Context initial = new InitialContext();
(14)            Object objRef = initial.lookup("personBean");
(15)
(16)            PersonHome home =
(17)                (PersonHome) PortableRemoteObject.narrow(
(18)                    objRef,
(19)                    PersonHome.class);
(20)            Person p1 = home.create(1911, 1, 1, "Alice", "streetA");
(21)            Person p2 = home.create(1922, 2, 2, "Bob", "streetB");
(22)            System.out.println("Alice's Age: " + p1.getAge());
(23)
(24)            System.out.println("Alice's primary key: " + p1.getPrimaryKey
(25)                ());
(26)            p1.remove();
(27)
(28)            Person p3 = home.findByPrimaryKey("Bob");
(29)            System.out.println(
(30)                "Bob's modified street (before modification): "
(31)                    + p3.getStreet());
(32)            p3.setStreet("streetC");
(33)            System.out.println(
(34)                "Bob's modified street (after modification): "
(35)                    + p3.getStreet());
(36)            System.out.println("also the other reference: " + p2.
(37)                getStreet());
(38)            System.out.println("Are both references the same Java
(39)                object: " + (p2==p3));
(40)            System.out.println("Are both references the same EJBObject
(41)                (calls isIdentical): " + p2.isIdentical(p3));
(42)        } catch (NamingException ne) {
(43)            ne.printStackTrace();
(44)        } catch (RemoteException re) {
(45)            re.printStackTrace();
(46)        } catch (CreateException ce) {
(47)            ce.printStackTrace();
(48)        } catch (RemoveException e) {
(49)            e.printStackTrace();
(50)        } catch (FinderException e) {
(51)            e.printStackTrace();
(52)        }
```



[Download des Beispiels](#)

Der Client ermittelt zunächst per JNDI eine Referenz auf die Bean, welche unter dem Namen *personBean* im Verzeichnisdienst registriert ist.

Die erhaltene generische Referenz wird durch Aufruf der statischen Methode [narrow](#) der Klasse [PortableRemoteObject](#) typsicher in eine Ausprägung der Home-Schnittstelle (*PersonHome*) konvertiert.

Der Aufruf der in dieser Schnittstelle durch den Anwender definierten create-Methode sorgt für die serverseitige Instanziierung der EJB, die als Ausprägung der Remote-Schnittstelle geliefert wird. Tatsächlich wird nicht das EJB-Objekt selbst durch den Methodenaufruf retourniert, sondern lediglich ein netzwerktransparenter Verweis darauf, der jedoch clientseitig einer lokalen Objektreferenz gleichgestellt verwendet werden kann.

Ferner wird serverseitig zur Kommunikation mit der EJB ein Home-Objekt erzeugt, welchem eine Stellvertreterrolle für den anfragenden Client zukommt.

Der Aufruf der durch die EJB zur Verfügung gestellten Methode erfolgt identisch zu dem einer Lokalen.

So dient der Aufruf der Methode create zur Erzeugung von serverseitig instanziiert und transparent persistierten EJB-Objekten sowie den lokalen Java-(Stellvertreter-)Objekten für den Zugriff darauf.

Der Aufruf von *getAge* zeigt die Nutzung einer in der Remote-Schnittstelle veröffentlichten Zugriffsmethode. Mit *getPrimaryKey* wird die, in der durch die Remote-Schnittstelle erweiterten Schnittstelle *EJBObject* angesiedelte, Operation zur Ermittlung des Primärschlüsselwertes eines EJB-Objektes aufgerufen.

Die Methode *remove* stellt dagegen eine durch die Home-Schnittstelle definierte Operation dar.

Durch den Aufruf dieser Methode auf dem durch *p1* referenzierten Objekt wird durch Ausführung der Beanmethode *ejbRemove* die die Bean serverseitig repräsentierenden Datenbankeinträge entfernt sowie der durch die Bean belegte Speicherbereich als frei markiert. Alle Versuche nach Aufruf dieser Methode auf der clientseitigen Hauptspeicherrepräsentation Wertänderungen durchzuführen führen daher zu einem Fehler.

Die Ermittlung von Referenzen auf existierende EJB-Objekte erfolgt durch die in der Remote-Schnittstelle definierte Methode *findByPrimaryKey*. Der EJB-Container stellt sicher, daß verschiedene Referenzen auf dasselbe EJB-Objekt synchronisiert in die Datenbank abgebildet werden, so daß keine Inkonsistenzen entstehen.

Für den Betrieb einer Enterprise Java Bean ist neben den bisher betrachteten Schnittstellen-Komponenten und der Realisierung der Bean selbst auch ein als *Deployment Descriptor* bezeichnetes XML-Konfigurationsfile notwendig, welches verschiedene Einstellungsdaten sowie die Schnittstellendaten enthält.

Beispiel 83 zeigt ein Beispiel hierfür:

Beispiel 83: Deployment Descriptor der Entity Bean

```
(1) <?xml version="1.0" encoding="UTF-8"?>
(2)
(3) <!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
(4)
(5) <ejb-jar>
(6)   <display-name>Ejb1</display-name>
(7)   <enterprise-beans>
(8)     <entity>
(9)       <display-name>PersonBean</display-name>
(10)      <ejb-name>PersonBean</ejb-name>
(11)      <home>PersonHome</home>
(12)      <remote>Person</remote>
(13)      <ejb-class>PersonBean</ejb-class>
(14)      <persistence-type>Bean</persistence-type>
(15)      <prim-key-class>java.lang.String</prim-key-class>
(16)      <reentrant>False</reentrant>
(17)      <security-identity>
(18)        <description></description>
(19)        <use-caller-identity></use-caller-identity>
(20)      </security-identity>
(21)    </entity>
(22)  </enterprise-beans>
(23)  <assembly-descriptor>
(24)    <method-permission>
(25)      <unchecked />
(26)    <method>
(27)      <ejb-name>PersonBean</ejb-name>
(28)      <method-impl>Remote</method-impl>
(29)      <method-name>getAge</method-name>
```

```

(30)         <method-params />
(31)     </method>
(32)     <method>
(33)         <ejb-name>PersonBean</ejb-name>
(34)         <method-intf>Home</method-intf>
(35)         <method-name>remove</method-name>
(36)         <method-params>
(37)             <method-param>javax.ejb.Handle</method-param>
(38)         </method-params>
(39)     </method>
(40)     <method>
(41)         <ejb-name>PersonBean</ejb-name>
(42)         <method-intf>Home</method-intf>
(43)         <method-name>getHomeHandle</method-name>
(44)         <method-params />
(45)     </method>
(46)     <method>
(47)         <ejb-name>PersonBean</ejb-name>
(48)         <method-intf>Remote</method-intf>
(49)         <method-name>isIdentical</method-name>
(50)         <method-params>
(51)             <method-param>javax.ejb.EJBObject</method-param>
(52)         </method-params>
(53)     </method>
(54)     <method>
(55)         <ejb-name>PersonBean</ejb-name>
(56)         <method-intf>Home</method-intf>
(57)         <method-name>remove</method-name>
(58)         <method-params>
(59)             <method-param>java.lang.Object</method-param>
(60)         </method-params>
(61)     </method>
(62)     <method>
(63)         <ejb-name>PersonBean</ejb-name>
(64)         <method-intf>Remote</method-intf>
(65)         <method-name>getHandle</method-name>
(66)         <method-params />
(67)     </method>
(68)     <method>
(69)         <ejb-name>PersonBean</ejb-name>
(70)         <method-intf>Home</method-intf>
(71)         <method-name>findByPrimaryKey</method-name>
(72)         <method-params>
(73)             <method-param>java.lang.String</method-param>
(74)         </method-params>
(75)     </method>
(76)     <method>
(77)         <ejb-name>PersonBean</ejb-name>
(78)         <method-intf>Home</method-intf>
(79)         <method-name>getEJBMetaData</method-name>
(80)         <method-params />
(81)     </method>
(82)     <method>
(83)         <ejb-name>PersonBean</ejb-name>
(84)         <method-intf>Remote</method-intf>
(85)         <method-name>getPrimaryKey</method-name>
(86)         <method-params />
(87)     </method>
(88)     <method>
(89)         <ejb-name>PersonBean</ejb-name>
(90)         <method-intf>Remote</method-intf>
(91)         <method-name>remove</method-name>
(92)         <method-params />
(93)     </method>
(94)     <method>
(95)         <ejb-name>PersonBean</ejb-name>
(96)         <method-intf>Remote</method-intf>
(97)         <method-name>getEJBHome</method-name>
(98)         <method-params />
(99)     </method>
(100) </method-permission>
(101) <container-transaction>

```



```

(102)     <method>
(103)         <ejb-name>PersonBean</ejb-name>
(104)         <method-intf>Home</method-intf>
(105)         <method-name>create</method-name>
(106)         <method-params>
(107)             <method-param>int</method-param>
(108)             <method-param>int</method-param>
(109)             <method-param>int</method-param>
(110)             <method-param>java.lang.String</method-param>
(111)             <method-param>java.lang.String</method-param>
(112)         </method-params>
(113)     </method>
(114)     <trans-attribute>Never</trans-attribute>
(115) </container-transaction>
(116) <container-transaction>
(117)     <method>
(118)         <ejb-name>PersonBean</ejb-name>
(119)         <method-intf>Remote</method-intf>
(120)         <method-name>setStreet</method-name>
(121)         <method-params>
(122)             <method-param>java.lang.String</method-param>
(123)         </method-params>
(124)     </method>
(125)     <trans-attribute>Never</trans-attribute>
(126) </container-transaction>
(127) <container-transaction>
(128)     <method>
(129)         <ejb-name>PersonBean</ejb-name>
(130)         <method-intf>Remote</method-intf>
(131)         <method-name>getStreet</method-name>
(132)         <method-params />
(133)     </method>
(134)     <trans-attribute>Never</trans-attribute>
(135) </container-transaction>
(136) <container-transaction>
(137)     <method>
(138)         <ejb-name>PersonBean</ejb-name>
(139)         <method-intf>Home</method-intf>
(140)         <method-name>remove</method-name>
(141)         <method-params>
(142)             <method-param>javax.ejb.Handle</method-param>
(143)         </method-params>
(144)     </method>
(145)     <trans-attribute>Never</trans-attribute>
(146) </container-transaction>
(147) <container-transaction>
(148)     <method>
(149)         <ejb-name>PersonBean</ejb-name>
(150)         <method-intf>Home</method-intf>
(151)         <method-name>remove</method-name>
(152)         <method-params>
(153)             <method-param>java.lang.Object</method-param>
(154)         </method-params>
(155)     </method>
(156)     <trans-attribute>Never</trans-attribute>
(157) </container-transaction>
(158) <container-transaction>
(159)     <method>
(160)         <ejb-name>PersonBean</ejb-name>
(161)         <method-intf>Remote</method-intf>
(162)         <method-name>remove</method-name>
(163)         <method-params />
(164)     </method>
(165)     <trans-attribute>Never</trans-attribute>
(166) </container-transaction>
(167) <container-transaction>
(168)     <method>
(169)         <ejb-name>PersonBean</ejb-name>
(170)         <method-intf>Remote</method-intf>
(171)         <method-name>getAge</method-name>
(172)         <method-params />
(173)     </method>

```



```

(174)      <trans-attribute>Never</trans-attribute>
(175)    </container-transaction>
(176)    <container-transaction>
(177)      <method>
(178)        <ejb-name>PersonBean</ejb-name>
(179)        <method-intf>Home</method-intf>
(180)        <method-name>findByPrimaryKey</method-name>
(181)        <method-params>
(182)          <method-param>java.lang.String</method-param>
(183)        </method-params>
(184)      </method>
(185)      <trans-attribute>Never</trans-attribute>
(186)    </container-transaction>
(187)  </assembly-descriptor>
(188)</ejb-jar>
(189)

```

[Download des Beispiels](#)

3.4 Web Services/SOAP

▲ 4 Server-seitige Präsentation und Sicherheitsaspekte

4.1 Java Server Pages (JSP)

Beim Einsatz von Servlets ist oftmals das Verhältnis zwischen berechneten Anteilen und statisch produzierten Ausgaben deutlich zu Gunsten der statischen Anteile verschoben. Gleichzeitig ist jedoch die Ausgabe statischen HTML- oder XML-Codes vergleichsweise mühsam und schreiaufwendig, wie Beispiel 72 zeigt.

Zur Abhilfe dies vermeidbaren Aufwende und damit zur Unterstützung von Applikationstypen in denen die dynamisch festzulegenden Anteile einer Ausgabe deutlich gegenüber den statischen zurückstehen sieht Sammlung der J2EE-Applikationstypen die *Java Server Pages* (JSP) vor.

Konzeptionell stellt eine Server Page eine --- gegenüber klassischem HTML/XML nahezu unveränderte --- Ausgabeseite dar, in die Anweisungen eingestreut werden können, welche zur Auslieferungszeit an den Client durch ihr Berechnungsergebnis ersetzt werden. Technisch ist der Ansatz auf Basis der existierenden Servlet-API realisiert und wird serverseitig auch transparent in ein solches übersetzt. Als einzige Einschränkung gegenüber der Mächtigkeit des Servletansatzes ist jedoch die Beschränkung auf die Verarbeitung von HTTP-POST-Anfragen zu sehen.

Beispiel 84 zeigt die Reformulierung der Servlet-basierten Umsetzung aus Beispiel 72 als JSP:

Beispiel 84: Einfache JSP

```

(1)<%@page import="java.util.Date"%>
(2)
(3)<html>
(4)   <head>
(5)     <title>Hello World!</title>
(6)   </head>
(7)   <body>
(8)     <h1>Hello World!</h1>
(9)     <p>Current date: <%= (new Date()).toString() %></p>
(10)  </body>
(11)</html>

```



[Download des Beispiels](#)

Augenfälligste Änderung gegenüber der Fassung als Servlet ist die direkte Darstellung der HTML-Ausgabeelemente. Die aktiven Inhalte werden durch Elemente, die mit dem Zeichen „%“ beginnen und mit „%“ enden eingeschlossen. Dazwischen befindet sich übersetzbarer Java-Quellcode.

Etwaige benötigte Bibliotheken werden im Java-üblichen import-Stil angegeben und zu Beginn der

JSP-Seite eingeführt.

Zu Beginn einer JSP-Seite können im Rahmen der page-Deklaration verschiedene seitenglobale Vereinbarungen getroffen werden:

- **language**
Wurde ursprünglich für eine (immernoch ausstehende) Unterstützung des JSP-Mechanismus durch verschiedene Programmiersprachen eingeführt. Vorgabegemäß hat das Attribut den Wert „Java“. Andere Werte sind gegenwärtig nicht definiert.
- **extends**
Erlaubt es die aus der JSP-Seite entstehende Servletklasse explizit als Spezialisierung der angegebenen Klasse zu definieren.
- **import**
Kommaseparierte Liste der verwendeten Bibliotheken.
- **session**
Legt fest, ob die JSP-Seite im Rahmen des HTTP-Sessionhandlings zustandsbehaftet verarbeitet wird.
- **buffer**
Erlaubt die Festlegung der innerhalb des übersetzten Servlets verwendeten Ausgabemechanismus. Vorgabegemäß und bei Angabe von none ist dies PrintWriter.
- **autoFlush**
Legt fest, daß gefüllte Ausgabepuffer automatisch an den anfragenden Client übermittelt werden.
- **isThreadSafe**
Weist dieses Attribut den Wert true auf, dann werden durch die Ausführungsumgebungen mehrere Anfragen zur selben Zeit an die JSP (bzw. das entstehende Servlet) übergeben.
- **info**
Möglichkeit zur Ablage informaler deskriptiver Information.
- **errorPage**
Referenz auf eine andere JSP, die zur Verarbeitung von Ausnahmeereignissen herangezogen wird.
- **isErrorPage**
Kennzeichnet ob eine JSP-Seite zur Fehlerbehandlung einer anderen Seite herangezogen werden kann und soll.

Die nähere Analyse des Quellecodes von Beispiel 84 sowie die Semantik der vorgestellten Attribute offenbart zwei unterschiedliche Codierungsformen für JSPs. Zunächst die durch <@ eingeleiteten vordefinierten **Direktiven**, welche im Rahmen des Servletübersetzungsprozesses in Java-Code transformiert werden. Diese Codeabbildung wird durch eine sog. *Tag Library*, einer Sammlung von Javacodefragmenten und -transformationsvorschriften zur dynamischen Erzeugung der Ausgabe, gesteuert.

Zu zweiten gestattet der JSP-Mechanismus die direkte Darstellung von Java-Quellcode, der zur Laufzeit direkt in ausführbaren Code umgesetzt wird im Rahmen der sog. **Scriptlets**. Diese durch <% eingeführten Codeanteile werden unverändert in das entstehende Servlet einkopiert.

Zusätzlich stellt das Beispiel, durch <%= eingeleitete, Ausdrücke (**Expressions**) dar, deren Resultatwert durch die Laufzeitumgebung in die an den Client übermittelte Seite eingefügt wird.

Im Beispiel nicht dargestellt sind Variablen- oder Konstantendeklarationen, welche auf die Einleitungssequenz <%! folgen.

Tag Libraries

Zur Strukturierung des Entwurfs und zur Wiederverwendung bereits einmal eingesetzter Codesequenzen für JSPs existiert mit dem Mechanismus der Tag Libraries die Möglichkeit anwenderdefiniert den Sprachumfang der Server Pages zu erweitern.

Einige häufig gebrauchte Aktionen sind bereits in der im Standardumfang enthaltenen Bibliothek versammelt:

- **useBean**
Zugriff auf eine Enterprise Java Bean.
- **setProperty**
Setzen einer Java-Property. Wird zumeist in Verbindung mit useBean verwendet um Bean-Eigenschaften festzulegen.
- **getProperty**
Auslesen einer Java-Property.
- **param**
Dient zur Parameterübergabe an andere Direktiven.
- **include**
Inklusion anderer Datenquellen.
- **forward**
Weiterleitung des Aufrufes der die JSP-Seite erreichte an andere Ressource.
- **plugin**
Einschluß eines Java-Applets in die erzeugte Ausgabe.

Beispiel 85 zeigt mit der import-Direktive die vordefinierte JSP-Aktion zur serverseitigen Inklusion existierender Dateien.

Beispiel 85: Importmechanismus für Server Pages



```
(1)<html>
(2)    <head>
(3)        <title>J. W. von Goethe: Faust I</title>
(4)    </head>
(5)    <body>
(6)        <h1>FAUST: EINE TRAGÖDIE</h1>
(7)        <h2>Zueignung.</h2>
(8)        <%@ include file="Zueignung.html" %>
(9)
(10)       <h2>Vorspiel auf dem Theater.</h2>
(11)       <%@ include file="Vorspiel.html" %>
(12)
(13)       <h2>Prolog im Himmel.</h2>
(14)       <%@ include file="Prolog.html" %>
(15)    </body>
(16)</html>
```

[Download des Beispiels](#)

Die Erzeugung eigener Tag Libraries geschieht durch Erstellung einer Java-Klasse, welche die Schnittstelle `javax.servlet.jsp.tagext.Tag` implementiert. Zumeist wird jedoch die Klasse `TagSupport` spezialisiert, die einige zusätzliche Methoden definiert, welche die Erstellung vereinfachen.

Beispiel 86 zeigt die Umsetzung des anwenderdefinierten Tags `hello` und Beispiel 87 seine Anwendung.

Beispiel 86: Erstellung einer Tag Library



```
(1)import java.io.IOException;
(2)import java.util.Date;
(3)
(4)import javax.servlet.jsp.JspTagException;
(5)import javax.servlet.jsp.tagext.TagSupport;
(6)
(7)public class HelloTag extends TagSupport {
(8)    public int doStartTag() throws JspTagException {
(9)        return EVAL_BODY_INCLUDE;
(10)    }
(11)
(12)    public int doEndTag() throws JspTagException {
(13)        try {
(14)            pageContext.getOut().write("Hello World!");
(15)            pageContext.getOut().write(
(16)                "current date: " + new Date().toString());
(17)        } catch (IOException ioe) {
(18)            ioe.printStackTrace();
(19)        }
(20)        return EVAL_PAGE;
(21)    }
(22)
(23)}
```

[Download des Beispiels](#)

Das Beispiel definiert Methoden für die beiden durch die Schnittstelle vorgegebenen Operationen `doStartTag` und `doEndTag`, die bei der Verarbeitung der entsprechenden Tag-Anteile aufgerufen werden. Der Rückgabewert der Methoden legt fest, daß die Inhalte des anwenderdefinierten Elements in der JSP-Seite in den Ausgabestrom übernommen werden (`EVAL_BODY_INCLUDE`) bzw. das mit der Verarbeitung der Seite fortgefahren werden soll (`EVAL_PAGE`).

Beispiel 87: Nutzung eines anwenderdefinierten Tags



```
(1)<%@ taglib uri="hello" prefix="example" %>
(2)<html>
(3)    <head>
(4)        <title>Hello World!</title>
(5)    </head>
(6)    <body>
(7)        <p>Statische Ausgabe ...</p>
(8)        <example:hello>
(9)        </example:hello>
(10)    </body>
(11)</html>
```

[Download des Beispiels](#)

Abschließende Bemerkung:

Der JSP-Mechanismus eignet sich sehr gut zur schnellen Erstellung leistungsfähiger Anwendungen einer gewissen Komplexität jedoch wirkt sich die physische Kopplung des Kontrollflusses an die erzeugte Ausgabe durch die Ablage in derselben Datei negativ auf die Skalierbarkeit hinsichtlich der betrachteten Problemgröße aus. Durch die mangelnde zentralisierte Verwaltbarkeit lassen sich komplexe Anwendungen oft nur noch schwer überschauen und testen.

Zusätzlich wirkt sich die erst zum Aufrufzeitpunkt serverseitig vorgenommene Übersetzung in ein Servlet nicht nur negativ auf die Laufzeit aus, sondern erschwert zusätzlich die Fehlersuche, da Syntaxfehler erst beim Aufruf entdeckt werden können.

4.2 Java Server Faces (JSF)

4.3 XML Transformationen mit XSLT

Die prominenteste Verwendung der [XPath](#)-Pfadausdrücke und eine der in jüngerer Zeit am weitesten beachteten XML-Vokabulare dürfte XSLT -- die Sprache der *XSL Transformations* -- sein. Sie wurde im Verlauf der Standardisierung der Stylesheetsprache XSL von dieser abgetrennt und seither durch eine eigene W3C-Arbeitsgruppe vorangetrieben.

In einem Satz zusammengefaßt stellt sie eine Turing-vollständige funktionale Programmiersprache zur Transformation wohlgeformter XML-Dokumente in beliebige Unicode-Streams -- und damit im speziellen wiederum in XML-Dokumente -- dar.

Der Begriff *Transformationen* bezeichnet hierbei die Selektion einzelner Bestandteile des Quelldokuments, deren Umordnung sowie die Ableitung neuer Inhalte aus den bereits bestehenden. Derzeit aktuell ist die [Recommendation zur Version 1.0](#) von Herbst 1999. Intern arbeitet das W3C bereits an der Nachfolgerversion XSLT v2.0, welche auch die absehbaren Entwicklungen des Umfeldes, wie XPath v2.0 oder XML-Schema, berücksichtigen wird.

Jedes XSLT-Stylesheet ist ein gültiges XML-Dokument, in dem alle Elemente der Sprache XSLT im Namensraum <http://www.w3.org/1999/XSL/Transform> plziert sind.

Üblicherweise wird der Namensraum an das Präfix `xsl` gebunden, welches allen XSLT-Sprachelementen explizit vorangestellt wird.

Das Wuzелеlement eines XSLT-Dokuments bildet der Knoten `stylesheet`. Alternativ kann auch `transform` angegeben werden. Zusätzlich verfügt jedes Stylesheet über ein Versionsattribut zur Bezeichnung der verwendeten XSLT-Version.

Das Beispiel zeigt ein minimales Stylesheet

Beispiel 88: Ein minimales Stylesheet



```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsl:transform version="1.0"
(3)    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"/>
```

[Download des Beispiels](#)

Angewendet auf das [Beispieldokument der Projektverwaltung](#) liefert es folgende Ausgabe:

```
(1)<?xml version="1.0" encoding="utf-8"?>
(2)Hans Hinterhuber
(3)Franz Xaver Obermüller
(4)IT-Kompetenz verschiedene Betriebssysteme und professionelle Programmierung
verschiedener Programmiersprachen
```

- (5)Entwickler von 1988-1990 Projektleiterfunktion von 1990-93 im X42-Projekt in Abteilung AB&C
 (6)Fritz Meier

Dies mag zunächst verwundern, definiert doch das Beispiel-Stylesheet keinerlei Transformationsregeln oder Ausgabefunktionen.

Das Ergebnis des minimal-Beispiels wird durch die [eingebauten Vorgaberegeln](#) erzeugt. Diese geben, sofern nicht anders angegeben alle [Character Information Items](#) unverändert aus.

Durch Überschreiben dieser Vorgaben und die Definition eigener Regeln lassen sich komplexe Transformationen auf dem Eingabedokument verwirklichen.

Transformationsschablonen

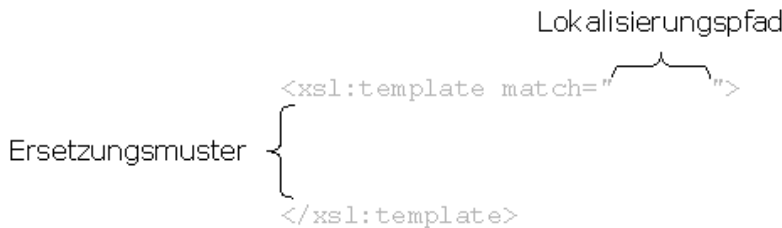


Abbildung 23: Aufbau einer Transformationsschablone (click on image to enlarge!)

Die Graphik zeigt den Aufbau einer Transformationsschablone. Ihre beiden Hauptbestandteile sind der *Lokalisierungspfad* und das *Ersetzungsmuster*.

Der Lokalisierungspfad (in der Spezifikation als *pattern* bezeichnet) liefert eine Knotenmenge. Als Syntax wird eine eingeschränkte Variante der Lokatorsprache XPath verwendet.

Augenfälligster Unterschied zur bisherigen Notation ist die Optionalität der descendant-Achse (zumeist zu `//` verkürzt) zur hierarchieebenenunabhängigen Lokalisierung eines Knotens.

Im Rumpf des Musters legt das Ersetzungsmuster diejenige Zeichenfolge fest, die statt jedem Element der lokalisierten Knotenmenge ausgegeben werden soll.

Das nachfolgende Transformationsheet liefert angewandt auf das [Projektverwaltungsbeispiel](#) dreimal die Ausgabe `Person gefunden!`; für jedes Auftreten des Knotens `Person` in der Eingabe.

Beispiel 89: Eine einfache Transformation

```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsl:transform version="1.0"
(3)  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
(4)
(5)    <xsl:template match="Person">
(6)      <xsl:text>Person gefunden!</xsl:text>
(7)    </xsl:template>
(8)</xsl:transform>
```



Download des Beispiels

Download der Ergebnisdatei

Innerhalb des Ersetzungsmusters können im allgemeinen beliebige Textsequenzen angegeben werden. Insbesondere ist die Verwendung wohlgeformter XML-Fragmente zugelassen.

Textsequenzen werden hierbei durch direktes Anschreiben, oder umschlossen durch das Element `xsl:text` definiert.

Anmerkung: Die Anforderungen an die Wohlgeformtheit sind hierbei auf die korrekte Terminierung der Elemente (damit einhergehend ihre korrekte Schachtelung), sowie die Quotierung der Attribute beschränkt.

Im folgenden Beispiel wird jedes Element des Typs `Person` durch ein leeres `Mitarbeiter`-Element ersetzt.

Das Beispiel erklärt auch die übliche Anwendungspraxis, alle XSLT-Elemente durch Namensraumpräfix zu qualifizieren, statt der -- weniger schreibaufwendigen -- Überschreibung des Vorgabennamensraumes. Würde der Vorgabennamensraum mit der Namensraum-URI der XSL-Transformations belegt, so befände sich auch jedes XML-Element und -Attribut innerhalb des Ersetzungsmusters in diesem Namensraum. Als Konsequenz würde der XSLT-Prozessor die Transformation wegen des Auftretens ungültiger (d.h. nicht in der XSLT-Sprache enthaltener) Elemente ablehnen. Bei Redefinition des Vorgabennamensraumes müßte daher für alle Elemente, die nicht Bestandteil von XSLT sind, eine explizite Namensraumdefinition im Element erfolgen, wodurch die Lesbarkeit stark herabgesetzt würde.

Beispiel 90: Erzeugung einer XML-Ausgabe



```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsl:transform version="1.0"
(3)  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
(4)
(5)<xsl:template match="Person">
(6)  <Mitarbeiter/>
(7)</xsl:template>
(8)
(9)</xsl:transform>
```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Erwartungsgemäß liefert das Beispiel dreifach das leere Element Mitarbeiter anstelle der Personen-Elemente der Eingabe.

Die bisher genutzte Form der Ersetzung ist jedoch für die praktische Anwendung in nur äußerst wenigen Fällen geeignet, da sie keine Übernahme von Daten der Eingabedatei in die Ausgabe erlaubt.

Dieses Manko wird durch das XSLT-Sprachelement value-of behoben.

Das Element value-of übernimmt als Teil des Ersetzungsmusters frei selektierbare Text-artige Informationsknoten aus dem Quelldokument. Die Lokalisierung der zu übernehmenden Knoten erfolgt durch XPath-Syntax innerhalb des select-Attributs.

Im folgenden Beispiel wird der Inhalt der Personen-Knoten in den neuen Knotentyp Mitarbeiter übernommen.

Beispiel 91: Übernahme bestehender Information aus dem Quelldokument



```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsl:transform version="1.0"
(3)  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
(4)
(5)<xsl:template match="Person">
(6)  <Mitarbeiter>
(7)    <xsl:value-of select="self:.*"/>
(8)  </Mitarbeiter>
(9)</xsl:template>
(10)
(11)</xsl:transform>
```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Das Resultat liefert jedoch nicht das Quelldokument, unter Umbenennung der Personen-Knoten in Mitarbeiter, sondern die dargestellte Textsequenz.

```
(1)<?xml version="1.0" encoding="utf-8"?>
(2)<Mitarbeiter>Hans Hinterhuber</Mitarbeiter>
(3)<Mitarbeiter>Franz Xaver Obermüller IT-Kompetenzverschiedene Betriebssysteme und
(4)professionelle
(5)Programmierung verschiedener Programmiersprachen Entwickler von 1988-1990
(6)Projektleiterfunktion
(7)von 1990-93 im X42-Projekt in Abteilung AB&amp;C</Mitarbeiter>
(8)<Mitarbeiter>Fritz Meier</Mitarbeiter>
```

Die Lösung liegt in der [Definition des value-of-Elements](#). Es konvertiert alle durch den im select-Attribut bezeichneten XPath lokalisierten Knoten in ihre Textrepräsentation. Im vorliegenden Beispiel ist dies der Inhalt der Knoten des Typs Person, der durch den Elementinhalt gebildet wird. Der Elementinhalt wird hierbei durch alle Kindknoten und deren Attribute gebildet.

Zur unveränderten Übernahme eines vollständigen Elements einschließlich der Auszeichnungssymbole wird das XSLT-Element [copy-of](#) angeboten.

Das nachfolgende Beispiel modifiziert das vorhergehend diskutierte Stylesheet dergestalt, daß für alle Personen-Knoten zunächst ein öffnender Mitarbeiter-Tag gesetzt wird. Im Rumpf des so begonnenen Elements werden durch das copy-of-Element alle Kindknoten der aktuellen Knotenmenge unverändert kopiert.

Als zusätzliche Veränderung gegenüber dem vorigen Beispiel fällt die Nutzung der child-Achse im select-Attribut des copy-of-Elements auf. Dies ist notwendig, da durch den Lokalisierungspfad der Schablone alle Personen-Knoten zu einer Ergebnisknotenmenge zusammengefaßt werden. Die

Anwendung der self-Achse innerhalb des copy-of-Elements würde daher auch die Personen-Knoten selbst übernehmen.

Zusammenfassend läßt sich die Funktionalität des Beispiels mit *Umbenennung aller Elemente des Typs Person in Mitarbeiter* wiedergeben.

Beispiel 92: Kopieren vollständiger Elementknoten

```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsl:transform version="1.0"
(3)  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
(4)
(5)<xsl:template match="//Person">
(6)  <Mitarbeiter>
(7)    <xsl:copy-of select="child::*"/>
(8)  </Mitarbeiter>
(9)</xsl:template>
(10)
(11)</xsl:transform>
```



[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Die vorgestellte Transformationsvorschrift ist hinreichend flexibel für Knoten des Typs Person auf beliebiger Hierarchiestufe des Eingabedokuments. Jedoch versagt sie bereits beim Versuch einen weiteren Transformationsschritt einzubauen, der auf einem der unverändert kopierten Kindelemente von Person operiert.

Das folgende Stylesheet stellt einen flexibleren Ansatz zur Umbenennung von einzelnen Knoten vor.

Gegenüber der vorhergehenden Fassung ist der Umfang um zwei weitere template-Elemente erweitert. Eines, das für alle Qualifikationsprofil-Knoten angewendet wird und eines für alle anderen Knoten. Der dort eingesetzte Lokalisierungspfad liefert als Ergebnismenge die Vereinigung aller Attributknoten (attribute::*) mit allen Knoten außer dem Wurzelknoten (node()-Funktion). Im Rumpf des Elements wird das copy-Element zur Kopie jedes Elements verwendet. Anders als das bisher herangezogene copy-of-Element übernimmt diese Variante ausschließlich das aktuelle Element in das Ergebnisdokument und läßt eventuell vorhandene Kindelemente unberücksichtigt. Das template-Element mit dem Lokalisierungspfad Qualifikationsprofil verfügt über kein Ersetzungsmuster. Es bewirkt damit die Unterdrückung des Teilbaumes unterhalb aller Knoten des Typs Qualifikationsprofil.

Die korrekte Anwendung der verschiedenen Schablonen wird durch den ausführenden XSLT-Prozessor gewährleistet, er ermittelt anhand der Lokalisierungspfade das best-passendste Template und bringt es zur Ausführung. Wenn, wie im vorliegenden Beispiel, mehrere Pfadausdrücke einen Knoten des Eingabedokuments lokalisieren, so wird das am weitesten spezifizierte Muster ausgewählt. Im untenstehenden Beispiel gilt dies für alle Elemente des Typs Person. Jeder dieser Knoten ist sowohl durch den XPath-Ausdruck der ersten Schablone als auch durch den allgemeineren Ausdruck node() zugänglich. Der explizite Pfad Person des ersten Lokalisierungsmusters ist jedoch gegenüber der Ergebnismenge der node-Funktion (deutlich) spezifischer.

Da jeder Knoten, außer denen des Typs Person und Qualifikationsprofil, unverändert in den Ausgabestrom übernommen werden soll, wird im Beispiel wieder ein copy-Element eingesetzt. Im Unterschied zum bisher verwendeten copy-of jedoch mit der Einschränkung, daß copy nur den aktuellen Knoten kopiert und eventuell existierende Kindknoten unberücksichtigt läßt. Dieser Vorgang wird auch als *shallow copy* bezeichnet.

Steuerung der Transformationsreihenfolge durch apply-templates-Elemente

Standardmäßig durchläuft ein XSLT-Prozessor den aus dem Eingabedokument erzeugten Baum ausgehend vom Wurzelknoten in Preorder Reihenfolge. Während des Traversierungsvorganges werden die zum jeweiligen Knoten „passenden“ Schablonen ausgewertet. „Passend“ deutet hierbei auf das Enthaltensein des besuchten Knotens in der Ergebnismenge eines XPath-Ausdrucks innerhalb eines match-Attributes hin.

Jedoch ist auch die anwenderdefinierte Beeinflussung der vorgegebenen Abarbeitungsreihenfolge möglich. Hierzu enthält das Beispiel zwei [apply-templates](#)-Elemente. Diese lösen einen Rekursionsschritt aus, dergestalt, daß an jeder Stelle, an der sich ein apply-templates-Aufruf findet, der Prozessor versucht, weitere passende Schablonen anhand der angegebenen Lokalisierungspfade zu ermitteln. Diese können an der gegebenen Stelle ausgewertet werden. Der Vorgang entspricht damit der Substitution in funktionalen Programmiersprachen.

Im vorliegenden Fall wird nach Ausgabe des öffnenden Tags Mitarbeiter -- nachdem ein

Personen-Knoten im Eingabedokument ermittelt wurde -- nach weiteren Knoten im Eingabedokument gesucht, zu denen Lokalisierungspfade in der XSLT-Transformationsvorschrift existieren. Dies ist für alle Attribute und Kindknoten von Person der Fall, da sie durch den Lokalisierungspfad `attribute::*|node()` zugänglich sind. So wird innerhalb des neu erzeugten Elements Mitarbeiter des Ausgabestroms das Ersetzungsmuster ausgeführt, das die Elemente und Attribute mit ihren Inhalten unverändert übernimmt.

Als Besonderheit nutzt das `apply-templates`-Element im „allgemeinen“ (dritten) template das Attribut `select`. Es erlaubt die anwenderdefinierte Steuerung der Menge, innerhalb der nach weiteren *passenden* Lokalisierungspfaden gesucht werden soll. Standardmäßig ist diese Menge mit allen dem aktuellen Element nachfolgenden (following-Achse) Elementknoten gefüllt. Im vorliegenden Beispiel wird sie durch den XPath des Attributwertes um alle Attributknoten erweitert.

Beispiel 93: Flexible Umbenennung und Löschung von Elementen

```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsl:transform version="1.0"
(3)  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
(4)
(5)<xsl:template match="Person">
(6)  <Mitarbeiter>
(7)    <xsl:apply-templates/>
(8)  </Mitarbeiter>
(9)</xsl:template>
(10)
(11)<xsl:template match="Qualifikationsprofil"/>
(12)
(13)<xsl:template match="attribute::*|node()">
(14)  <xsl:copy>
(15)    <xsl:apply-templates select="attribute::*|node()"/>
(16)  </xsl:copy>
(17)</xsl:template>
(18)
(19)</xsl:transform>
```



[Download des Beispiels](#)

[Download der Ergebnisdatei](#)



Übung 3: Verfolgung der Aufrufreihenfolge

Lassen Sie sich mit von einem XSLT-Prozessor die Aufrufreihenfolge der einzelnen Templates ausgeben.

Beispielsweise mit dem Prozessor [Xalan](#) aus dem Apache-Projekt.

Benannte Ersetzungsschablonen und Parameterübergabe

Neben den Lokationspfad-gesteuerten Schablonen kann der Anwender auch benannte Schablonen, ohne `match`-Attribut, definieren. Diese werden während der Abarbeitung des Eingabebaumes nicht berücksichtigt, sondern müssen manuell durch `call-template` aufgerufen werden.

Konzeptionell entsprechen sie Funktionsaufrufen herkömmlicher Programmiersprachen.

(In [Spezifikation nachschlagen](#)).

Die Definition erfolgt analog den bisher genutzten Schablonen, mit der Ausnahme, daß statt dem `match`-Attribut ein eindeutiger Name (Attribute name) angegeben wird.

Als neuer Freiheitsgrad beim nun notwendigen manuellen Aufruf tritt die Möglichkeit der Parameterübergabe hinzu. Als Parameter können beliebige Dokumentbestandteile als Knotenmenge, Ergebnisse von Funktionsausdrücken oder Konstanten übergeben werden. Eine Parameterrückgabe ist nicht möglich, sie wird durch den Anteil der Schablone an der Ausgabe realisiert.

Die Aufrufsyntax lautet:

```
(1)<xsl:call-template name="QName">
(2)  <!-- Content: xsl:with-param* -->
(3)</xsl:call-template>
(4)<xsl:with-param name="QName" select="eingeschränkter XPath-Ausdruck">
(5)  <!-- Content: template -->
(6)</xsl:with-param>
```

Anmerkung: Wie in allen funktionalen Sprachen kommt den Funktionen dieses Typs besondere Bedeutung, als Ausgangspunkt rekursiver Aufrufe, zu.

Die Vorgabe-Transformationsregeln

Das [einführende Beispiel dieses Kapitels](#) griff bereits auf die in den XSLT-Prozessor „eingebauten“ [Standard-Transformationsvorschriften](#) (*built-in templates*) zurück.

So lautet die Definition, welche für alle Text- und Attributknoten der Eingabe den Inhalt in die Ausgabe kopiert:

```
(1)<xsl:template match="text()|@">
(2)    <xsl:value-of select="."/>
(3)</xsl:template>
```

Ferner existiert ein template zur rekursiven Abarbeitung des Eingabebaumes, welches immer dann Anwendung findet, wenn sich keine spezialisiertere Transformationsvorschrift findet.

```
(1)<xsl:template match="*/">
(2)    <xsl:apply-templates/>
(3)</xsl:template>
```

Ergänzt wird diese Zusammenstellung durch eine Schablone zur Eliminierung von Namensraum- und Kommentarknoten.

Elemente der Ablaufsteuerung

Ähnlich zu herkömmlichen Programmiersprachen bietet auch XSLT Sprachmittel zur Selektion und bedingten Verarbeitung abhängig von den Eingabedaten.

So erlaubt das if-Element die Bearbeitung der umschlossenen Elemente nur, wenn die im test-Attribut formulierte Boole'sche Bedingung wahr ist.

([In Spezifikation nachschlagen](#))

Die Syntax lautet:

```
<xsl:if test="Boole'scher Ausdruck">    <!-- Content: template --> </xsl:if>
```

Das Beispiel zeigt die Nutzung des if-Elements. Verfügt ein Knoten des Typs Person über mehr als einen Kindknoten des Typs Vorname so wird der Inhalt des if-Elements abgearbeitet, der durch das XSLT-Element message während des Transformationsvorganges eine Bildschirmausgabe erzeugt. Diese gibt zunächst den Inhalt des Nachnamen Knotens gefolgt von einem statischen Text aus. Angewandt auf das Beispieldokument liefert es auf Kommandozeile die Ausgabe: Obermüller hat mehr als einen Vornamen!. Der Inhalt des Eingabedokuments wird unverändert kopiert.

Beispiel 94: Bedingte Verarbeitung durch Verwendung des if-Elements

```
(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsl:transform version="1.0"
(3)    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
(4)
(5)<xsl:template match="Person">
(6)    <Mitarbeiter>
(7)        <xsl:if test="count(Vorname)>1">
(8)            <xsl:message><xsl:value-of select="Nachname"/> hat mehr als
einen Vornamen!</xsl:message>
(9)        </xsl:if>
(10)        <xsl:apply-templates/>
(11)    </Mitarbeiter>
(12)</xsl:template>
(13)
(14)<xsl:template match="Qualifikationsprofil"/>
(15)
(16)<xsl:template match="attribute::*|node()">
(17)    <xsl:copy>
(18)        <xsl:apply-templates select="attribute::*|node()"/>
(19)    </xsl:copy>
(20)</xsl:template>
(21)
(22)</xsl:transform>
```



[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

In Erweiterung der simplen Selektion bildet choose die Möglichkeiten einer Mehrfachselektion, oder

einer simplen if-then-else-Struktur ab.

([In Spezifikation nachschlagen](#))

Die Syntax lautet:

```
(1)<xsl:choose>
(2)    <!-- Content: (xsl:when+, xsl:otherwise?) -->
(3)</xsl:choose>
(4)<xsl:when test="Boole'scher Ausdruck">
(5)    <!-- Content: template -->
(6)</xsl:when>
(7)<xsl:otherwise>
(8)    <!-- Content: template -->
(9)</xsl:otherwise>
```

Das Beispiel zeigt die Transformation des [Beispieldokuments](#) in eine XHTML-Ausgabe.

Hierbei wird zunächst der XHTML-Dokumentrahmen bei Auftreten des Dokumentknotens (Suchmuster /) erzeugt. Nach dem öffnenden XHTML-Rumpfelement body werden innerhalb des Quelldokuments wahlfrei weitere, auf eines der angegebenen Suchmuster passende, Knoten gesucht (apply-templates-Aufruf).

Bei Auftreten des Knotens ProjektVerwaltung wird -- nach einigen Überschriftszeilen -- der Kopf einer XHTML-Tabelle, bestehend aus den Elementen table und der Kopfzeile (eingeschlossen durch tr), erzeugt. Den Rumpf der Tabelle schreibt ein anderes Template. Dieses wird jedoch nicht direkt aufgerufen, sondern der Prozeß der Ermittlung neuer „passender“ Knoten neu initiiert; jedoch diesmal, durch das select-Attribut des apply-templates-Elements, nur auf Knoten des Typs Person beschränkt.

Die Ersetzungsregel für Person speichert zunächst den Inhalt des Attributs PersID in einer Variable. Anschließend werden die Tabellenelemente der in der Ersetzungsregel für ProjektVerwaltung geöffneten Tabelle geschrieben. Hierzu werden nacheinander die Ersetzungsmuster für Knoten des Typs Vorname bzw. Nachname, die Kindknoten des aktuellen Knotens sind (die PersID des aktuellen Personen-Knotens findet sich in der zuvor belegten Variable), aktiviert.

Ein zweites Tabellenelement enthält einen XHTML-Hyperlink zu den durch den Mitarbeiter bearbeiteten Projekten. Als Sprungziel wird hierbei der Inhalt des Attributs mitarbeitInProjekt eingetragen.

Das Ersetzungsmuster Vorname übernimmt durch value-of die in einen Zeichenkettenwert gewandelten Inhalte des Elements. Zusätzlich existiert ein zweites Ersetzungsmuster für Knoten des Typs Vorname, das jedoch durch ein Prädikat nur auf das zweite und alle folgenden Elemente dieses Typs angewendet wird. Es gibt vor der Übernahme des Elementinhaltes ein Leerzeichen aus. Das Element Nachname bettet den Elementinhalt in eine benannte Sprungreferenz ein. Zur Erzeugung der dokumentweiten eindeutigen Benennung wird die Funktion generate-id herangezogen die für jedes Element des Information Sets des Eingabedokuments einen eindeutigen Bezeichner liefert.

Nach Abschluß der Tabellengenerierung im durch den Lokalisierungspfad ProjektVerwaltung gekennzeichneten Template werden durch den Aufruf des benannten Ersetzungsmusters wasteSpace 25 Leerzeilen, jeweils gefüllt mit der Zeichenkette „...“, erzeugt. Als Besonderheit ist in diesem Muster sehr deutlich die Nutzung der Rekursion zur Modellierung einer Schleife zu sehen. Zum Abschluß wird nochmals eine Tabelle, nun mit den Mitarbeitern und ihren Projekten, erzeugt.

Beispiel 95: Erzeugung eines XHTML-Reports

```
(1)<?xml version="1.0"?>
(2)<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
(3)
(4)<xsl:output method="xml" encoding="ISO-8859-1" omit-xml-declaration="no"
indent="yes"
(5)          xmlns="http://www.w3.org/1999/xhtml" />
(6)
(7)<xsl:template match="/">
(8)    <html>
(9)      <head>
(10)        <title>XML-Vorlesung Sommersemester 2001 --
Projektverwaltung</title>
(11)      </head>
(12)      <body>
(13)        <xsl:apply-templates/>
(14)      </body>
(15)    </html>
(16)</xsl:template>
(17)
(18)<xsl:template match="ProjektVerwaltung">
(19)  <center><h1>Projektverwaltung</h1></center>
```

```

(20)
(21) <h2>Projekte</h2>
(22) <table border="1">
(23)     <tr>
(24)         <td><b>Projektnummer</b></td>
(25)         <td><b>Projektleiter</b></td>
(26)     </tr>
(27)     <xsl:apply-templates select="Projekt"/>
(28) </table>
(29)
(30) <pre>
(31)     <xsl:call-template name="wasteSpace">
(32)         <xsl:with-param name="maxLines">25</xsl:with-param>
(33)         <xsl:with-param name="curLines">0</xsl:with-param>
(34)     </xsl:call-template>
(35) </pre>
(36)
(37) <h2>Mitarbeiter</h2>
(38) <table border="1">
(39)     <tr>
(40)         <td><b>Name</b></td>
(41)         <td><b>Projekt</b></td>
(42)     </tr>
(43)     <xsl:apply-templates select="Person"/>
(44) </table>
(45)</xsl:template>
(46)
(47) <xsl:template name="wasteSpace">
(48)     <xsl:param name="maxLines"/>
(49)     <xsl:param name="curLines"/>
(50)     <xsl:if test="$curLines < $maxLines">
(51)<xsl:text>...
(52)</xsl:text>
(53)         <xsl:call-template name="wasteSpace">
(54)             <xsl:with-param name="maxLines">25</xsl:with-param>
(55)             <xsl:with-param name="curLines"><xsl:value-of
select="$curLines + 1"/></xsl:with-param>
(56)         </xsl:call-template>
(57)     </xsl:if>
(58) </xsl:template>
(59)
(60)<xsl:template match="Projekt">
(61)     <xsl:variable name="prjLeiter" select="@Projektleiter"/>
(62)     <tr>
(63)         <td>
(64)             <xsl:element name="a">
(65)                 <xsl:attribute name="name"><xsl:value-of
select="@ID"/></xsl:attribute>
(66)                 <xsl:value-of select="@ID"/>
(67)             </xsl:element>
(68)         </td>
(69)         <td>
(70)             <xsl:element name="a">
(71)                 <xsl:attribute name="href">#<xsl:value-of
select="generate-id(//Person[@PersID=$prjLeiter]/Nachname)"/></xsl:attribute>
(72)                 <xsl:value-of select="//Person[@PersID=$prjLeiter]/
Nachname"/>
(73)             </xsl:element>
(74)         </td>
(75)     </tr>
(76)</xsl:template>
(77)
(78)<xsl:template match="Person">
(79)     <xsl:variable name="persNr" select="@PersID"/>
(80)     <tr>
(81)         <td><xsl:apply-templates select="Vorname[parent::Person/@PersID=
$persNr]"/>&#160;
(82)         <xsl:apply-templates select="Nachname[parent::Person/
@PersID=$persNr]"/></td>
(83)         <td>
(84)             <xsl:element name="a">
(85)                 <xsl:attribute name="href">#<xsl:value-of

```



```

select="@mitarbeitInProjekt"/></xsl:attribute>
(86)          <xsl:value-of select="@mitarbeitInProjekt"/>
(87)          </xsl:element>
(88)      </td>
(89)  </tr>
(90)</xsl:template>
(91)
(92)<xsl:template match="Vorname">
(93)  <xsl:value-of select="."/>
(94)</xsl:template>
(95)
(96)<xsl:template match="Vorname[position() > 1]">
(97)  <xsl:text>&#160;</xsl:text><xsl:value-of select="."/>
(98)</xsl:template>
(99)
(100)<xsl:template match="Nachname">
(101)  <xsl:element name="a">
(102)    <xsl:attribute name="name"><xsl:value-of select="generate-id()"/></
xsl:attribute>
(103)    <xsl:value-of select="."/>
(104)  </xsl:element>
(105)</xsl:template>
(106)
(107)<xsl:template match="text()"/>
(108)
(109)</xsl:stylesheet>

```

[Download des Beispiels](#)

[Download der Ergebnisdatei](#)

Die nachfolgende XSLT-Transformation ermittelt zu jedem beliebigen Element- und Attributknoten (Muster: * bzw. @*) die zugehörige Namensraum-URI (Funktion [namespace-uri](#)) und gibt sie gemeinsam mit dem Namen des Knotens (Funktion [name](#)) in einer XML-formatierten Ausgabe aus. Zur Neuselektion aller weiteren Element- und Attributknoten wird apply-templates mit dem Musterausdruck @*|node() ausgeführt, um in die Prüfung nach weiteren passenden Knoten (node ()) auch explizit alle beliebigen Attribute (@*) einzubeziehen. Standardmäßig ermittelt apply-templates weitere passende Knoten nur innerhalb der Elemente eines Dokuments.

Beispiel 96: Ausgabe Namensräume jedes Elements und Attributs eines beliebigen XML-Dokuments

```

(1)<?xml version="1.0" encoding="UTF-8"?>
(2)<xsl:transform version="1.0"
(3)  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
(4)
(5)<xsl:output method="xml" encoding="ISO-8859-1" omit-xml-declaration="no"
indent="yes" />
(6)
(7)<xsl:template match="*">
(8)  <xsl:element name="element">
(9)    <xsl:attribute name="name"><xsl:value-of select="name()"/></xsl:
attribute>
(10)    <xsl:attribute name="namespace"><xsl:value-of select="namespace-uri
()"/></xsl:attribute>
(11)    <xsl:apply-templates select="@*|node()"/>
(12)  </xsl:element>
(13)</xsl:template>
(14)
(15)<xsl:template match="@*">
(16)  <xsl:element name="attribute">
(17)    <xsl:attribute name="name"><xsl:value-of select="name()"/></xsl:
attribute>
(18)    <xsl:attribute name="namespace"><xsl:value-of select="namespace-uri
()"/></xsl:attribute>
(19)    <xsl:apply-templates select="@*|node()"/>
(20)  </xsl:element>
(21)</xsl:template>
(22)
(23)<xsl:template match="text()"/>
(24)
(25)</xsl:transform>

```



Download des Beispiels

Web-Referenzen 10: Weiterführende Links



- [XSLT v1.0 Spezifikation](#)
- [XSLT @ jeckle.de](#)
- [Holman, K. G.: What is XSLT?](#)
- [Saxon -- ein freier XSLT-Prozessor](#)
- [Apache Xalan -- ein Open Source XSLT-Prozessor](#)

Übung 4: Textuelle Aufbereitung der Baumstruktur eines XML-Dokuments

Schreiben Sie eine XSLT-Transformation, die es erlaubt die Elementstruktur beliebiger wohlgeformter XML-Dokumente in Form eines „ASCII-Baums“ (siehe Beispiel) am Bildschirm auszugeben.

Beispiel:

Eingabe:



```
(1)<root>
(2)  <childX>
(3)      <childY1/>
(4)      <childY2/>
(5)  </childX>
(6)  <childX2/>
(7)</root>
```

Ausgabe:

```
(1)root
(2)+--childX
(3)  +--childY1
(4)  +--childY2
(5)+--childX2
```

▲ Definitionsverzeichnis

[e-Business](#)

[Gültigkeit hinsichtlich eines Schemas](#)

[Lokalisierungsschritt](#)

[Namensräume](#)

[Namensraumidentifikation](#)

[Namensraumvererbung](#)

[Wohlgeformtes XML-Dokument](#)

[XML Dokument](#)

[XML Information Set](#)

[XML-Prozessor](#)

[XML-Sprache](#)

▲ Schlagwortverzeichnis

[B2B](#)

[B2C](#)

[B2E](#)

[Business Logik](#)

[Business-to-Business](#)

[Business-to-Customer](#)

[Business-to-Employee](#)

[e-Business](#)

[e-Commerce](#)

[entfernte Methodenauf](#)

[e-tailing](#)

[Internettechniken](#)
[Java Database Connectivity](#)
[JDBC](#)
[JDBC-Treiber](#)
[lazy activation](#)
[ODBC](#)
[Open Database Connectivity](#)
[Personalisierung](#)
[Remote Method Invocation](#)
[Skeleton](#)
[SQL/CLI](#)
[SQL Call Level Interface](#)
[Stub](#)
[Typ 1 Treiber](#)
[Typ 2 Treiber](#)
[Typ 3 Treiber](#)
[Typ 4 Treiber](#)

Abbildungsverzeichnis

[Dimensionen des e-Business](#)
[Schichtenmodell moderner e-Business Applikationen](#)
[JDBC-Treibertypen](#)
[Zentrale JDBC-Klassen der Java-API](#)
[JDBC-Geschwindigkeitsvergleich](#)
[Verteilungsstruktur einer RMI-Applikation](#)
[Struktur der Servlet-API](#)
[Aufrufstruktur einer zustandslosen EJB](#)

Verzeichnis der Beispiele

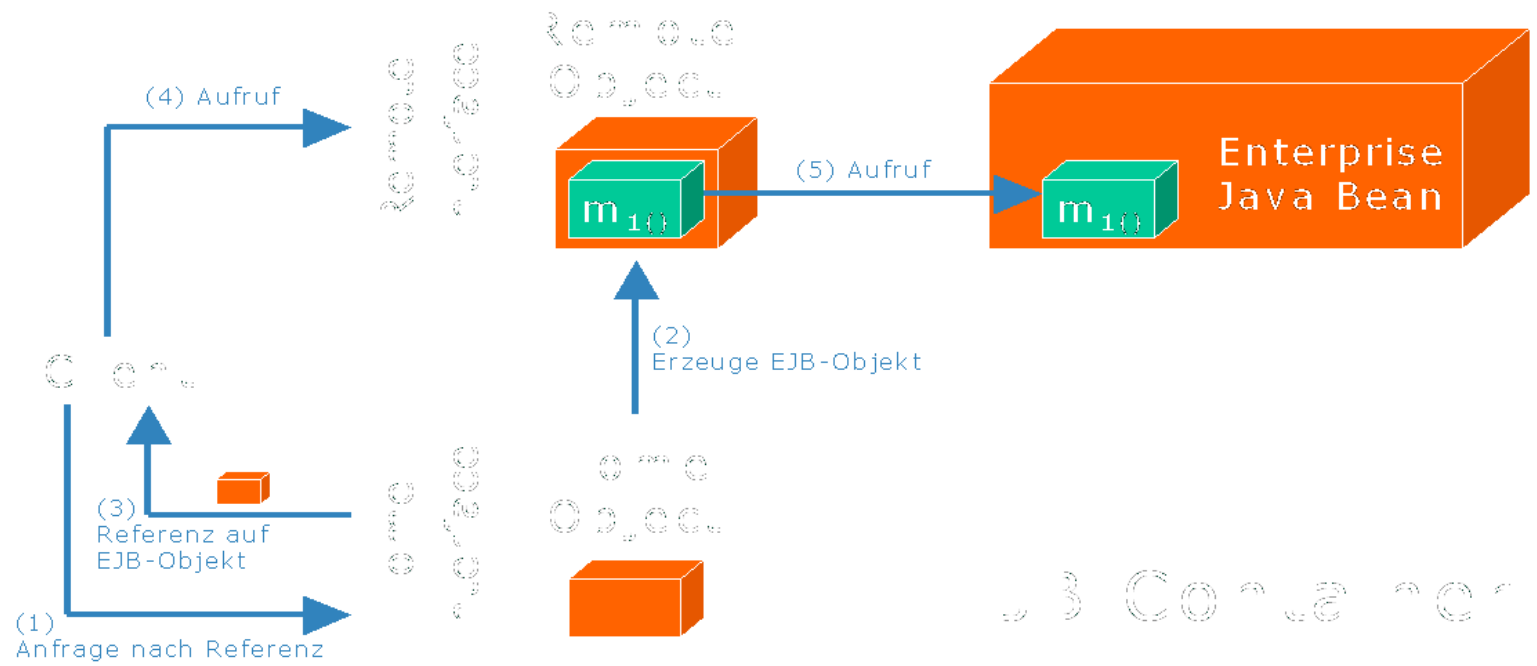
Service provided by [Mario Jeckle](#)

Generated: 2004-01-08T22:57:08+01:00

 [Feedback](#)  [SiteMap](#)

 [This page's original location: http://www.jeckle.de/vorlesung/eBusinessEng/script.html](#)

 [RDF description for this page](#)



```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface SayHelloHome extends EJBHome {
    public SayHello create() throws RemoteException, CreateException;
}
```



```
import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface SayHello extends EJBObject {
    public String sayHello(String name) throws RemoteException;
}
```

```
import java.rmi.RemoteException;
import java.util.Date;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class HelloWorldBean implements SessionBean {
    public HelloWorldBean() {
    }

    public String sayHello(String name) {
        return ("Hello " + name + " it is now " + new Date().toString());
    }

    public void setSessionContext(SessionContext arg0)
        throws EJBException, RemoteException {
    }

    public void ejbRemove() throws EJBException, RemoteException {
    }

    public void ejbCreate() throws EJBException {
    }

    public void ejbActivate() throws EJBException, RemoteException {
    }

    public void ejbPassivate() throws EJBException, RemoteException {
    }
}
```

java.io

Interface Serializable

All Known Subinterfaces:

[AdapterActivator](#), [Attribute](#), [Attribute](#), [Attributes](#), [BindingIterator](#), [ClientRequestInfo](#),
[ClientRequestInterceptor](#), [Codec](#), [CodecFactory](#), [Control](#), [Current](#), [Current](#), [Current](#),
[CustomValue](#), [DataInputStream](#), [DataOutputStream](#), [DHPrivateKey](#), [DHPublicKey](#),
[DocAttribute](#), [DomainManager](#), [DSAPrivateKey](#), [DSAPublicKey](#), [DynAny](#), [DynAnyFactory](#),
[DynArray](#), [DynEnum](#), [DynFixed](#), [DynSequence](#), [DynStruct](#), [DynUnion](#), [DynValue](#),
[DynValueBox](#), [DynValueCommon](#), [ExtendedRequest](#), [ExtendedResponse](#), [Externalizable](#),
[IdAssignmentPolicy](#), [IDLEntity](#), [IDLType](#), [IdUniquenessPolicy](#), [ImplicitActivationPolicy](#),
[Interceptor](#), [IORInfo](#), [IORInterceptor](#), [IRObject](#), [Key](#), [LifespanPolicy](#), [Name](#), [NamingContext](#),
[NamingContextExt](#), [ORBInitializer](#), [ORBInitInfo](#), [PBEKey](#), [POA](#), [POAManager](#), [Policy](#),
[PolicyFactory](#), [PrintJobAttribute](#), [PrintRequestAttribute](#), [PrintServiceAttribute](#), [PrivateKey](#),
[PublicKey](#), [RemoteRef](#), [RequestInfo](#), [RequestProcessingPolicy](#),
[RSAMultiPrimePrivateCrtKey](#), [RSAPrivateCrtKey](#), [RSAPrivateKey](#), [RSAPublicKey](#),
[RunTime](#), [SecretKey](#), [ServantActivator](#), [ServantLocator](#), [ServantManager](#),
[ServantRetentionPolicy](#), [ServerRef](#), [ServerRequestInfo](#), [ServerRequestInterceptor](#),
[StreamableValue](#), [SupportedValuesAttribute](#), [ThreadPolicy](#), [UnsolicitedNotification](#),
[ValueBase](#)

public interface **Serializable**

Serializability of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

To allow subtypes of non-serializable classes to be serialized, the subtype may assume responsibility for saving and restoring the state of the supertype's public, protected, and (if accessible) package fields. The subtype may assume this responsibility only if the class it extends has an accessible no-arg constructor to initialize the class's state. It is an error to declare a class Serializable if this is not the case. The error will be detected at runtime.

During deserialization, the fields of non-serializable classes will be initialized using the public or

protected no-arg constructor of the class. A no-arg constructor must be accessible to the subclass that is serializable. The fields of serializable subclasses will be restored from the stream.

When traversing a graph, an object may be encountered that does not support the `Serializable` interface. In this case the `NotSerializableException` will be thrown and will identify the class of the non-serializable object.

Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

```
private void writeObject(java.io.ObjectOutputStream out)
    throws IOException
private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

The `writeObject` method is responsible for writing the state of the object for its particular class so that the corresponding `readObject` method can restore it. The default mechanism for saving the Object's fields can be invoked by calling `out.defaultWriteObject`. The method does not need to concern itself with the state belonging to its superclasses or subclasses. State is saved by writing the individual fields to the `ObjectOutputStream` using the `writeObject` method or by using the methods for primitive data types supported by `DataOutput`.

The `readObject` method is responsible for reading from the stream and restoring the classes fields. It may call `in.defaultReadObject` to invoke the default mechanism for restoring the object's non-static and non-transient fields. The `defaultReadObject` method uses information in the stream to assign the fields of the object saved in the stream with the correspondingly named fields in the current object. This handles the case when the class has evolved to add new fields. The method does not need to concern itself with the state belonging to its superclasses or subclasses. State is saved by writing the individual fields to the `ObjectOutputStream` using the `writeObject` method or by using the methods for primitive data types supported by `DataOutput`.

Serializable classes that need to designate an alternative object to be used when writing an object to the stream should implement this special method with the exact signature:

```
ANY-ACCESS-MODIFIER Object writeReplace() throws
ObjectStreamException;
```

This `writeReplace` method is invoked by serialization if the method exists and it would be accessible from a method defined within the class of the object being serialized. Thus, the method can have private, protected and package-private access. Subclass access to this method follows java accessibility rules.

Classes that need to designate a replacement when an instance of it is read from the stream should

implement this special method with the exact signature.

ANY-ACCESS-MODIFIER Object readResolve() throws
ObjectStreamException;

This readResolve method follows the same invocation rules and accessibility rules as writeReplace.

Since:

JDK1.1

See Also:

[ObjectOutputStream](#), [ObjectInputStream](#), [ObjectOutput](#), [ObjectInput](#),
[Externalizable](#)

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*JavaTM 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) [DETAIL: FIELD](#) | [CONSTR](#) | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

java.rmi

Interface Remote

All Known Subinterfaces:

[ActivationInstantiator](#), [ActivationMonitor](#), [ActivationSystem](#), [Activator](#), [DGC](#), [Registry](#)

All Known Implementing Classes:

[Remote_Stub](#), [ActivationGroup](#), [ActivationGroup_Stub](#), [RemoteObject](#)public interface **Remote**

The **Remote** interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a "remote interface", an interface that extends `java.rmi.Remote` are available remotely.

Implementation classes can implement any number of remote interfaces and can extend other remote implementation classes. RMI provides some convenience classes that remote object implementations can extend which facilitate remote object creation. These classes are `java.rmi.server.UnicastRemoteObject` and `java.rmi.activation.Activatable`.

For complete details on RMI, see the [RMI Specification](#) which describes the RMI API and system.

Since:

JDK1.1

See Also:

[UnicastRemoteObject](#), [Activatable](#)[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)SUMMARY: [INNER](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

javax.ejb

Interface EJBObject

All Superinterfaces:

java.rmi.Remote

public interface **EJBObject**

extends java.rmi.Remote

The EJBObject interface is extended by all enterprise Beans' remote interfaces. An enterprise Bean's remote interface provides the remote client view of an EJB object. An enterprise Bean's remote interface defines the business methods callable by a remote client.

The remote interface must extend the javax.ejb.EJBObject interface, and define the enterprise Bean specific business methods.

The enterprise Bean's remote interface is defined by the enterprise Bean provider and implemented by the enterprise Bean container.

Method Summary

EJBHome	getEJBHome () Obtain the enterprise Bean's remote home interface.
Handle	getHandle () Obtain a handle for the EJB object.
java. lang. Object	getPrimaryKey () Obtain the primary key of the EJB object.
boolean	isIdentical (EJBObject obj) Test if a given EJB object is identical to the invoked EJB object.
void	remove () Remove the EJB object.

Method Detail

getEJBHome

```
public EJBHome getEJBHome()  
           throws java.rmi.RemoteException
```

Obtain the enterprise Bean's remote home interface. The remote home interface defines the enterprise Bean's create, finder, remove, and home business methods.

Returns:

A reference to the enterprise Bean's home interface.

Throws:

[java.rmi.RemoteException](#) - Thrown when the method failed due to a system-level failure.

getPrimaryKey

```
public java.lang.Object getPrimaryKey()  
           throws java.rmi.RemoteException
```

Obtain the primary key of the EJB object.

This method can be called on an entity bean. An attempt to invoke this method on a session bean will result in [RemoteException](#).

Returns:

The EJB object's primary key.

remove

```
public void remove()  
           throws java.rmi.RemoteException,  
                  RemoveException
```

Remove the EJB object.

Throws:

[java.rmi.RemoteException](#) - Thrown when the method failed due to a system-level failure.

[RemoveException](#) - The enterprise Bean or the container does not allow destruction of the object.

getHandle

```
public Handle getHandle()  
    throws java.rmi.RemoteException
```

Obtain a handle for the EJB object. The handle can be used at later time to re-obtain a reference to the EJB object, possibly in a different Java Virtual Machine.

Returns:

A handle for the EJB object.

Throws:

[java.rmi.RemoteException](#) - Thrown when the method failed due to a system-level failure.

isIdentical

```
public boolean isIdentical(EJBObject obj)  
    throws java.rmi.RemoteException
```

Test if a given EJB object is identical to the invoked EJB object.

Parameters:

obj - An object to test for identity with the invoked object.

Returns:

True if the given EJB object is identical to the invoked object, false otherwise.

Throws:

[java.rmi.RemoteException](#) - Thrown when the method failed due to a system-level failure.

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [INNER](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;

public class CallHelloWorldBean {
    public static void main(String[] args) {
        try {
            Context initial = new InitialContext();
            Object objRef = initial.lookup("helloBean");

            SayHelloHome home =
                (SayHelloHome) PortableRemoteObject.narrow(
                    objRef,
                    SayHelloHome.class);
            SayHello sh = home.create();

            System.out.println(sh.sayHello("Mario"));

        } catch (NamingException ne) {
            ne.printStackTrace();
        } catch (RemoteException re) {
            re.printStackTrace();
        } catch (CreateException ce) {
            ce.printStackTrace();
        }
    }
}
```

javax.rmi

Class PortableRemoteObject



public class **PortableRemoteObject**
extends [Object](#)

Server implementation objects may either inherit from javax.rmi.PortableRemoteObject or they may implement a remote interface and then use the exportObject method to register themselves as a server object. The toStub method takes a server implementation and returns a stub that can be used to access that server object. The connect method makes a Remote object ready for remote communication. The unexportObject method is used to deregister a server object, allowing it to become available for garbage collection. The narrow method takes an object reference or abstract interface type and attempts to narrow it to conform to the given interface. If the operation is successful the result will be an object of the specified type, otherwise an exception will be thrown.

Constructor Summary	
protected	PortableRemoteObject () Initializes the object by calling exportObject(this).

Method Summary	
static void	connect (Remote target, Remote source) Makes a Remote object ready for remote communication.
static void	exportObject (Remote obj) Makes a server object ready to receive remote calls.
static Object	narrow (Object narrowFrom, Class narrowTo) Checks to ensure that an object of a remote or abstract interface type can be cast to a desired type.

static Remote	toStub (Remote obj) Returns a stub for the given server object.
static void	unexportObject (Remote obj) Deregisters a server object from the runtime, allowing the object to become available for garbage collection.

Methods inherited from class java.lang.[Object](#)

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

Constructor Detail

PortableRemoteObject

protected **PortableRemoteObject**()
throws [RemoteException](#)

Initializes the object by calling `exportObject(this)`.

Throws:

[RemoteException](#) - if export fails.

Method Detail

exportObject

public static void **exportObject**([Remote](#) obj)
throws [RemoteException](#)

Makes a server object ready to receive remote calls. Note that subclasses of `PortableRemoteObject` do not need to call this method, as it is called by the constructor.

Parameters:

obj - the server object to export.

Throws:

[RemoteException](#) - if export fails.

toStub

```
public static Remote toStub(Remote obj)
        throws NoSuchObjectException
```

Returns a stub for the given server object.

Parameters:

obj - the server object for which a stub is required. Must either be a subclass of `PortableRemoteObject` or have been previously the target of a call to [exportObject](#) (`java.rmi.Remote`).

Returns:

the most derived stub for the object.

Throws:

[NoSuchObjectException](#) - if a stub cannot be located for the given server object.

unexportObject

```
public static void unexportObject(Remote obj)
        throws NoSuchObjectException
```

Deregisters a server object from the runtime, allowing the object to become available for garbage collection.

Parameters:

obj - the object to unexport.

Throws:

[NoSuchObjectException](#) - if the remote object is not currently exported.

narrow

```
public static Object narrow(Object narrowFrom,
        Class narrowTo)
        throws ClassCastException
```

Checks to ensure that an object of a remote or abstract interface type can be cast to a desired type.

Parameters:

`narrowFrom` - the object to check.

`narrowTo` - the desired type.

Returns:

an object which can be cast to the desired type.

Throws:

[ClassCastException](#) - if `narrowFrom` cannot be cast to `narrowTo`.

connect

```
public static void connect(Remote target,
                           Remote source)
    throws RemoteException
```

Makes a Remote object ready for remote communication. This normally happens implicitly when the object is sent or received as an argument on a remote method call, but in some circumstances it is useful to perform this action by making an explicit call. See the `Stub#connect` method for more information.

Parameters:

`target` - the object to connect.

`source` - a previously connected object.

Throws:

[RemoteException](#) - if `source` is not connected or if `target` is already connected to a different ORB than `source`.

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*Java™ 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;

public interface PersonHome extends EJBHome {
    public Person create(int year, int month, int day, String name, String street) throws
CreateException, RemoteException;
    public Person findByPrimaryKey(String name) throws FinderException, RemoteException;
}
```



```
import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Person extends EJBObject {
    public int getAge() throws RemoteException;
    public String getStreet() throws RemoteException;
    public void setStreet(String street) throws RemoteException;
}
```

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)SUMMARY: [INNER](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

javax.ejb

Interface EntityBean**All Superinterfaces:**[EnterpriseBean](#), java.io.Serializablepublic interface **EntityBean**extends [EnterpriseBean](#)

The EntityBean interface is implemented by every entity enterprise Bean class. The container uses the EntityBean methods to notify the enterprise Bean instances of the instance's life cycle events.

Method Summary

void	ejbActivate() A container invokes this method when the instance is taken out of the pool of available instances to become associated with a specific EJB object.
void	ejbLoad() A container invokes this method to instruct the instance to synchronize its state by loading it state from the underlying database.
void	ejbPassivate() A container invokes this method on an instance before the instance becomes disassociated with a specific EJB object.
void	ejbRemove() A container invokes this method before it removes the EJB object that is currently associated with the instance.
void	ejbStore() A container invokes this method to instruct the instance to synchronize its state by storing it to the underlying database.
void	setEntityContext(EntityContext ctx) Set the associated entity context.

void	<u>unsetEntityContext()</u>

Unset the associated entity context.

Method Detail

setEntityContext

```
public void setEntityContext(EntityContext ctx)
                        throws EJBException,
                        java.rmi.RemoteException
```

Set the associated entity context. The container invokes this method on an instance after the instance has been created.

This method is called in an unspecified transaction context.

Parameters:

`ctx` - An `EntityContext` interface for the instance. The instance should store the reference to the context in an instance variable.

Throws:

[EJBException](#) - Thrown by the method to indicate a failure caused by a system-level error.

`java.rmi.RemoteException` - This exception is defined in the method signature to provide backward compatibility for enterprise beans written for the EJB 1.0 specification. Enterprise beans written for the EJB 1.1 specification should throw the `javax.ejb.EJBException` instead of this exception. Enterprise beans written for the EJB2.0 and higher specifications must throw the `javax.ejb.EJBException` instead of this exception.

unsetEntityContext

```
public void unsetEntityContext()
                        throws EJBException,
                        java.rmi.RemoteException
```

Unset the associated entity context. The container calls this method before removing the instance.

This is the last method that the container invokes on the instance. The Java garbage collector will eventually invoke the `finalize()` method on the instance.

This method is called in an unspecified transaction context.

Throws:

[EJBException](#) - Thrown by the method to indicate a failure caused by a system-level error.

`java.rmi.RemoteException` - This exception is defined in the method signature to provide backward compatibility for enterprise beans written for the EJB 1.0 specification. Enterprise beans written for the EJB 1.1 specification should throw the `javax.ejb.EJBException` instead of this exception. Enterprise beans written for the EJB2.0 and higher specifications must throw the `javax.ejb.EJBException` instead of this exception.

ejbRemove

```
public void ejbRemove()  
    throws RemoveException,  
           EJBException,  
           java.rmi.RemoteException
```

A container invokes this method before it removes the EJB object that is currently associated with the instance. This method is invoked when a client invokes a remove operation on the enterprise Bean's home interface or the EJB object's remote interface. This method transitions the instance from the ready state to the pool of available instances.

This method is called in the transaction context of the remove operation.

Throws:

[RemoveException](#) - The enterprise Bean does not allow destruction of the object.

[EJBException](#) - Thrown by the method to indicate a failure caused by a system-level error.

`java.rmi.RemoteException` - This exception is defined in the method signature to provide backward compatibility for enterprise beans written for the EJB 1.0 specification. Enterprise beans written for the EJB 1.1 specification should throw the `javax.ejb.EJBException` instead of this exception. Enterprise beans written for the EJB2.0 and higher specifications must throw the `javax.ejb.EJBException` instead of this exception.

ejbActivate

```
public void ejbActivate()  
    throws EJBException,  
           java.rmi.RemoteException
```

A container invokes this method when the instance is taken out of the pool of available instances to become associated with a specific EJB object. This method transitions the instance to the ready state.

This method executes in an unspecified transaction context.

Throws:

[EJBException](#) - Thrown by the method to indicate a failure caused by a system-level error.

`java.rmi.RemoteException` - This exception is defined in the method signature to provide backward compatibility for enterprise beans written for the EJB 1.0 specification. Enterprise beans written for the EJB 1.1 specification should throw the `javax.ejb.EJBException` instead of this exception. Enterprise beans written for the EJB2.0 and higher specifications must throw the `javax.ejb.EJBException` instead of this exception.

ejbPassivate

```
public void ejbPassivate()
           throws EJBException,
                  java.rmi.RemoteException
```

A container invokes this method on an instance before the instance becomes disassociated with a specific EJB object. After this method completes, the container will place the instance into the pool of available instances.

This method executes in an unspecified transaction context.

Throws:

[EJBException](#) - Thrown by the method to indicate a failure caused by a system-level error.

`java.rmi.RemoteException` - This exception is defined in the method signature to provide backward compatibility for enterprise beans written for the EJB 1.0 specification. Enterprise beans written for the EJB 1.1 specification should throw the `javax.ejb.EJBException` instead of this exception. Enterprise beans written for the EJB2.0 and higher specifications must throw the `javax.ejb.EJBException` instead of this exception.

ejbLoad

```
public void ejbLoad()
```

throws [EJBException](#),
`java.rmi.RemoteException`

A container invokes this method to instruct the instance to synchronize its state by loading it state from the underlying database.

This method always executes in the transaction context determined by the value of the transaction attribute in the deployment descriptor.

Throws:

[EJBException](#) - Thrown by the method to indicate a failure caused by a system-level error.

`java.rmi.RemoteException` - This exception is defined in the method signature to provide backward compatibility for enterprise beans written for the EJB 1.0 specification. Enterprise beans written for the EJB 1.1 specification should throw the `javax.ejb.EJBException` instead of this exception. Enterprise beans written for the EJB2.0 and higher specifications must throw the `javax.ejb.EJBException` instead of this exception.

ejbStore

```
public void ejbStore()  
    throws EJBException,  
           java.rmi.RemoteException
```

A container invokes this method to instruct the instance to synchronize its state by storing it to the underlying database.

This method always executes in the transaction context determined by the value of the transaction attribute in the deployment descriptor.

Throws:

[EJBException](#) - Thrown by the method to indicate a failure caused by a system-level error.

`java.rmi.RemoteException` - This exception is defined in the method signature to provide backward compatibility for enterprise beans written for the EJB 1.0 specification. Enterprise beans written for the EJB 1.1 specification should throw the `javax.ejb.EJBException` instead of this exception. Enterprise beans written for the EJB2.0 and higher specifications must throw the `javax.ejb.EJBException` instead of this exception.

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: INNER | FIELD | CONSTR | [METHOD](#)

DETAIL: FIELD | CONSTR | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)SUMMARY: [INNER](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

javax.ejb

Interface EnterpriseBean

All Superinterfaces:[java.io.Serializable](#)**All Known Subinterfaces:**[EntityBean](#), [MessageDrivenBean](#), [SessionBean](#)public interface **EnterpriseBean**extends [java.io.Serializable](#)

The EnterpriseBean interface must be implemented by every enterprise Bean class. It is a common superinterface for the SessionBean, EntityBean and MessageDrivenBean interfaces.

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)[PREV CLASS](#) [NEXT CLASS](#)[FRAMES](#) [NO FRAMES](#)SUMMARY: [INNER](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

```
import java.rmi.RemoteException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Calendar;
import java.util.GregorianCalendar;

import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.FinderException;
import javax.ejb.RemoveException;

public class PersonBean implements EntityBean {
    //persistent fields
    private String name;
    private GregorianCalendar birthdate;
    private String street;

    public PersonBean() {
    }

    //methods which are part of the remote interface
    public int getAge() {
        return (
            new GregorianCalendar().get(Calendar.YEAR)
            - birthdate.get(Calendar.YEAR));
    }
    public String getStreet() {
        return street;
    }
    public void setStreet(String street) throws RemoteException {
        if (street.length() <= 30) {
            this.street = street;
        } else {
            throw new RemoteException("cannot update street");
        }
    }
    // -----
    public String ejbCreate(
        int year,
        int month,
```

```
int day,
String name,
String street)
throws CreateException {
if (year > 1900
    && month >= 1
    && month <= 12
    && day >= 1
    && day <= 31
    && name.length() <= 20
    && street.length() <= 30) {

    this.name = name;
    this.birthdate = new GregorianCalendar(year, month, day);
    this.street = street;
    Statement stmt = getStatement();
    String s =
        new String(
            "INSERT INTO PERSON VALUES ("
                + name
                + ", "
                + birthdate.get(Calendar.YEAR)
                + " - "
                + birthdate.get(Calendar.MONTH)
                + " - "
                + birthdate.get(Calendar.DATE)
                + ", "
                + ""
                + street
                + ""
                + ");");
    try {
        stmt.executeUpdate(s);
    } catch (SQLException e) {
        e.printStackTrace();
    }
} else {
    throw new CreateException("Invalid values supplied");
}
return name;
}

public void ejbPostCreate(
    int year,
    int month,
    int day,
    String name,
```

```
String street) {
}
// -----
public int ejbHomeGetAge() {
    return 0;
}

public String ejbHomeGetStreet() {
    return new String();
}

public void ejbHomeSetStreet(String street) {
}

public Statement getStatement() {
    try {
        Class.forName("com.mysql.jdbc.Driver");
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    Connection con = null;
    try {
        con =
            (Connection) DriverManager.getConnection(
                "jdbc:mysql://10.0.0.1/Address/",
                "mario",
                "thePassword");
    } catch (SQLException e1) {
        e1.printStackTrace();
    }
    try {
        return ((Statement) con.createStatement());
    } catch (SQLException e2) {
        e2.printStackTrace();
    }
    return null; //never gets here
}

// -----
public void setEntityContext(EntityContext ectx)
    throws EJBException, RemoteException {
}

public void unsetEntityContext() throws EJBException, RemoteException {
}
```

```
public void ejbRemove()
    throws RemoveException, EJBException, RemoteException {
    Statement stmt = getStatement();
    String s = new String("DELETE FROM PERSON WHERE Name=" + name + ";"");
    try {
        stmt.executeUpdate(s);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

public void ejbActivate() throws EJBException, RemoteException {
}

public void ejbPassivate() throws EJBException, RemoteException {
}

public void ejbLoad() throws EJBException, RemoteException {
}

public void ejbStore() throws EJBException, RemoteException {
    Statement stmt = getStatement();
    String s =
        new String(
            "UPDATE PERSON SET Name="
                + name
                + ", BDATE="
                + birthdate.get(Calendar.YEAR)
                + "-"
                + birthdate.get(Calendar.MONTH)
                + "-"
                + birthdate.get(Calendar.DATE)
                + ", Street = "
                + street
                + " WHERE Name = "
                + name
                + ";"");
    try {
        stmt.executeUpdate(s);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

public String ejbFindByPrimaryKey(String name) throws FinderException {
    Statement stmt = getStatement();
```

```
String s =
    new String("SELECT Name FROM PERSON WHERE Name=" + name + ";");
try {
    ResultSet rs = stmt.executeQuery(s);
    rs.first();
    return rs.getString("Name");
} catch (SQLException e) {
    e.printStackTrace();
}
return null; //never gets here
}
```

java.util

Class GregorianCalendar

[java.lang.Object](#)└ [java.util.Calendar](#)└ **java.util.GregorianCalendar**

All Implemented Interfaces:

[Cloneable](#), [Serializable](#)public class **GregorianCalendar**extends [Calendar](#)

GregorianCalendar is a concrete subclass of [Calendar](#) and provides the standard calendar used by most of the world.

The standard (Gregorian) calendar has 2 eras, BC and AD.

This implementation handles a single discontinuity, which corresponds by default to the date the Gregorian calendar was instituted (October 15, 1582 in some countries, later in others). The cutover date may be changed by the caller by calling `setGregorianCalendarChange()`.

Historically, in those countries which adopted the Gregorian calendar first, October 4, 1582 was thus followed by October 15, 1582. This calendar models this correctly. Before the Gregorian cutover, **GregorianCalendar** implements the Julian calendar. The only difference between the Gregorian and the Julian calendar is the leap year rule. The Julian calendar specifies leap years every four years, whereas the Gregorian calendar omits century years which are not divisible by 400.

GregorianCalendar implements *proleptic* Gregorian and Julian calendars. That is, dates are computed by extrapolating the current rules indefinitely far backward and forward in time. As a result, **GregorianCalendar** may be used for all years to generate meaningful and consistent results. However, dates obtained using **GregorianCalendar** are historically accurate only from March 1, 4 AD onward, when modern Julian calendar rules were adopted. Before this date, leap year rules were applied irregularly, and before 45 BC the Julian calendar did not even exist.

Prior to the institution of the Gregorian calendar, New Year's Day was March 25. To avoid confusion,

this calendar always uses January 1. A manual adjustment may be made if desired for dates that are prior to the Gregorian changeover and which fall between January 1 and March 24.

Values calculated for the `WEEK_OF_YEAR` field range from 1 to 53. Week 1 for a year is the earliest seven day period starting on `getFirstDayOfWeek()` that contains at least `getMinimalDaysInFirstWeek()` days from that year. It thus depends on the values of `getMinimalDaysInFirstWeek()`, `getFirstDayOfWeek()`, and the day of the week of January 1. Weeks between week 1 of one year and week 1 of the following year are numbered sequentially from 2 to 52 or 53 (as needed).

For example, January 1, 1998 was a Thursday. If `getFirstDayOfWeek()` is `MONDAY` and `getMinimalDaysInFirstWeek()` is 4 (these are the values reflecting ISO 8601 and many national standards), then week 1 of 1998 starts on December 29, 1997, and ends on January 4, 1998. If, however, `getFirstDayOfWeek()` is `SUNDAY`, then week 1 of 1998 starts on January 4, 1998, and ends on January 10, 1998; the first three days of 1998 then are part of week 53 of 1997.

Values calculated for the `WEEK_OF_MONTH` field range from 0 to 6. Week 1 of a month (the days with `WEEK_OF_MONTH = 1`) is the earliest set of at least `getMinimalDaysInFirstWeek()` contiguous days in that month, ending on the day before `getFirstDayOfWeek()`. Unlike week 1 of a year, week 1 of a month may be shorter than 7 days, need not start on `getFirstDayOfWeek()`, and will not include days of the previous month. Days of a month before week 1 have a `WEEK_OF_MONTH` of 0.

For example, if `getFirstDayOfWeek()` is `SUNDAY` and `getMinimalDaysInFirstWeek()` is 4, then the first week of January 1998 is Sunday, January 4 through Saturday, January 10. These days have a `WEEK_OF_MONTH` of 1. Thursday, January 1 through Saturday, January 3 have a `WEEK_OF_MONTH` of 0. If `getMinimalDaysInFirstWeek()` is changed to 3, then January 1 through January 3 have a `WEEK_OF_MONTH` of 1.

Example:

```
// get the supported ids for GMT-08:00 (Pacific Standard
Time)
String[] ids = TimeZone.getAvailableIDs(-8 * 60 * 60 *
1000);
// if no ids were returned, something is wrong. get out.
if (ids.length == 0)
    System.exit(0);

// begin output
System.out.println("Current Time");

// create a Pacific Standard Time time zone
SimpleTimeZone pdt = new SimpleTimeZone(-8 * 60 * 60 *
1000, ids[0]);
```

```

    // set up rules for daylight savings time
    pdt.setStartRule(Calendar.APRIL, 1, Calendar.SUNDAY, 2 *
60 * 60 * 1000);
    pdt.setEndRule(Calendar.OCTOBER, -1, Calendar.SUNDAY, 2
* 60 * 60 * 1000);

    // create a GregorianCalendar with the Pacific Daylight
time zone
    // and the current date and time
    Calendar calendar = new GregorianCalendar(pdt);
    Date trialTime = new Date();
    calendar.setTime(trialTime);

    // print out a bunch of interesting things
    System.out.println("ERA: " + calendar.get(Calendar.ERA));
    System.out.println("YEAR: " + calendar.get(Calendar.
YEAR));
    System.out.println("MONTH: " + calendar.get(Calendar.
MONTH));
    System.out.println("WEEK_OF_YEAR: " + calendar.get
(Calendar.WEEK_OF_YEAR));
    System.out.println("WEEK_OF_MONTH: " + calendar.get
(Calendar.WEEK_OF_MONTH));
    System.out.println("DATE: " + calendar.get(Calendar.
DATE));
    System.out.println("DAY_OF_MONTH: " + calendar.get
(Calendar.DAY_OF_MONTH));
    System.out.println("DAY_OF_YEAR: " + calendar.get
(Calendar.DAY_OF_YEAR));
    System.out.println("DAY_OF_WEEK: " + calendar.get
(Calendar.DAY_OF_WEEK));
    System.out.println("DAY_OF_WEEK_IN_MONTH: "
+ calendar.get(Calendar.
DAY_OF_WEEK_IN_MONTH));
    System.out.println("AM_PM: " + calendar.get(Calendar.
AM_PM));
    System.out.println("HOUR: " + calendar.get(Calendar.
HOUR));
    System.out.println("HOUR_OF_DAY: " + calendar.get
(Calendar.HOUR_OF_DAY));
    System.out.println("MINUTE: " + calendar.get(Calendar.
MINUTE));
    System.out.println("SECOND: " + calendar.get(Calendar.
SECOND));
    System.out.println("MILLISECOND: " + calendar.get

```



```

(Calendar.MILLISECOND));
    System.out.println("ZONE_OFFSET: "
                       + (calendar.get(Calendar.ZONE_OFFSET)/
(60*60*1000)));
    System.out.println("DST_OFFSET: "
                       + (calendar.get(Calendar.DST_OFFSET)/
(60*60*1000)));

    System.out.println("Current Time, with hour reset to 3");
    calendar.clear(Calendar.HOUR_OF_DAY); // so doesn't
override
    calendar.set(Calendar.HOUR, 3);
    System.out.println("ERA: " + calendar.get(Calendar.ERA));
    System.out.println("YEAR: " + calendar.get(Calendar.
YEAR));
    System.out.println("MONTH: " + calendar.get(Calendar.
MONTH));
    System.out.println("WEEK_OF_YEAR: " + calendar.get
(Calendar.WEEK_OF_YEAR));
    System.out.println("WEEK_OF_MONTH: " + calendar.get
(Calendar.WEEK_OF_MONTH));
    System.out.println("DATE: " + calendar.get(Calendar.
DATE));
    System.out.println("DAY_OF_MONTH: " + calendar.get
(Calendar.DAY_OF_MONTH));
    System.out.println("DAY_OF_YEAR: " + calendar.get
(Calendar.DAY_OF_YEAR));
    System.out.println("DAY_OF_WEEK: " + calendar.get
(Calendar.DAY_OF_WEEK));
    System.out.println("DAY_OF_WEEK_IN_MONTH: "
                       + calendar.get(Calendar.
DAY_OF_WEEK_IN_MONTH));
    System.out.println("AM_PM: " + calendar.get(Calendar.
AM_PM));
    System.out.println("HOUR: " + calendar.get(Calendar.
HOUR));
    System.out.println("HOUR_OF_DAY: " + calendar.get
(Calendar.HOUR_OF_DAY));
    System.out.println("MINUTE: " + calendar.get(Calendar.
MINUTE));
    System.out.println("SECOND: " + calendar.get(Calendar.
SECOND));
    System.out.println("MILLISECOND: " + calendar.get
(Calendar.MILLISECOND));
    System.out.println("ZONE_OFFSET: "
                       + (calendar.get(Calendar.ZONE_OFFSET)/

```

```
(60*60*1000)); // in hours
    System.out.println("DST_OFFSET: "
        + (calendar.get(Calendar.DST_OFFSET)/
            (60*60*1000))); // in hours
```

Since:

JDK1.1

See Also:
[Calendar](#), [TimeZone](#), [Serialized Form](#)

Field Summary

static int	AD Value of the ERA field indicating the common era (Anno Domini), also known as CE.
static int	BC Value of the ERA field indicating the period before the common era (before Christ), also known as BCE.

Fields inherited from class java.util.[Calendar](#)

[AM](#), [AM_PM](#), [APRIL](#), [areFieldsSet](#), [AUGUST](#), [DATE](#), [DAY_OF_MONTH](#), [DAY_OF_WEEK](#), [DAY_OF_WEEK_IN_MONTH](#), [DAY_OF_YEAR](#), [DECEMBER](#), [DST_OFFSET](#), [ERA](#), [FEBRUARY](#), [FIELD_COUNT](#), [fields](#), [FRIDAY](#), [HOUR](#), [HOUR_OF_DAY](#), [isSet](#), [isTimeSet](#), [JANUARY](#), [JULY](#), [JUNE](#), [MARCH](#), [MAY](#), [MILLISECOND](#), [MINUTE](#), [MONDAY](#), [MONTH](#), [NOVEMBER](#), [OCTOBER](#), [PM](#), [SATURDAY](#), [SECOND](#), [SEPTEMBER](#), [SUNDAY](#), [THURSDAY](#), [time](#), [TUESDAY](#), [UNDECIMBER](#), [WEDNESDAY](#), [WEEK_OF_MONTH](#), [WEEK_OF_YEAR](#), [YEAR](#), [ZONE_OFFSET](#)

Constructor Summary

[GregorianCalendar](#)()

Constructs a default GregorianCalendar using the current time in the default time zone with the default locale.

[GregorianCalendar](#)(int year, int month, int date)

Constructs a GregorianCalendar with the given date set in the default time zone with the default locale.

[GregorianCalendar](#)(int year, int month, int date, int hour, int minute)

Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale.

[GregorianCalendar](#)(int year, int month, int date, int hour, int minute, int second)

Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale.

[GregorianCalendar](#)([Locale](#) aLocale)

Constructs a GregorianCalendar based on the current time in the default time zone with the given locale.

[GregorianCalendar](#)([TimeZone](#) zone)

Constructs a GregorianCalendar based on the current time in the given time zone with the default locale.

[GregorianCalendar](#)([TimeZone](#) zone, [Locale](#) aLocale)

Constructs a GregorianCalendar based on the current time in the given time zone with the given locale.

Method Summary

void	add (int field, int amount) Adds the specified (signed) amount of time to the given time field, based on the calendar's rules.
protected void	computeFields () Converts UTC as milliseconds to time field values.
protected void	computeTime () Overrides Calendar Converts time field values to UTC as milliseconds.
boolean	equals (Object obj) Compares this GregorianCalendar to an object reference.
int	getActualMaximum (int field) Return the maximum value that this field could have, given the current date.
int	getActualMinimum (int field) Return the minimum value that this field could have, given the current date.
int	getGreatestMinimum (int field) Returns highest minimum value for the given field if varies.
Date	getGregorianCalendarChange () Gets the GregorianCalendar change date.

int	<code>getLeastMaximum</code> (int field) Returns lowest maximum value for the given field if varies.
int	<code>getMaximum</code> (int field) Returns maximum value for the given field.
int	<code>getMinimum</code> (int field) Returns minimum value for the given field.
int	<code>hashCode</code> () Override hashCode.
boolean	<code>isLeapYear</code> (int year) Determines if the given year is a leap year.
void	<code>roll</code> (int field, boolean up) Adds or subtracts (up/down) a single unit of time on the given time field without changing larger fields.
void	<code>roll</code> (int field, int amount) Add to field a signed amount without changing larger fields.
void	<code>setGregorianChange</code> (<code>Date</code> date) Sets the GregorianCalendar change date.

Methods inherited from class java.util.[Calendar](#)

[after](#), [before](#), [clear](#), [clear](#), [clone](#), [complete](#), [get](#), [getAvailableLocales](#), [getFirstDayOfWeek](#), [getInstance](#), [getInstance](#), [getInstance](#), [getInstance](#), [getMinimalDaysInFirstWeek](#), [getTime](#), [getTimeInMillis](#), [getTimeZone](#), [internalGet](#), [isLenient](#), [isSet](#), [set](#), [set](#), [set](#), [set](#), [setFirstDayOfWeek](#), [setLenient](#), [setMinimalDaysInFirstWeek](#), [setTime](#), [setTimeInMillis](#), [setTimeZone](#), [toString](#)

Methods inherited from class java.lang.[Object](#)

[finalize](#), [getClass](#), [notify](#), [notifyAll](#), [wait](#), [wait](#), [wait](#)

Field Detail

BC

public static final int **BC**

Value of the **ERA** field indicating the period before the common era (before Christ), also known as BCE. The sequence of years at the transition from BC to AD is ..., 2 BC, 1 BC, 1 AD, 2 AD,...

See Also:

[Calendar.ERA](#), [Constant Field Values](#)

AD

```
public static final int AD
```

Value of the **ERA** field indicating the common era (Anno Domini), also known as CE. The sequence of years at the transition from BC to AD is ..., 2 BC, 1 BC, 1 AD, 2 AD,...

See Also:

[Calendar.ERA](#), [Constant Field Values](#)

Constructor Detail

GregorianCalendar

```
public GregorianCalendar()
```

Constructs a default **GregorianCalendar** using the current time in the default time zone with the default locale.

GregorianCalendar

```
public GregorianCalendar(TimeZone zone)
```

Constructs a **GregorianCalendar** based on the current time in the given time zone with the default locale.

Parameters:

zone - the given time zone.

GregorianCalendar

```
public GregorianCalendar(Locale aLocale)
```

Constructs a `GregorianCalendar` based on the current time in the default time zone with the given locale.

Parameters:

`aLocale` - the given locale.

GregorianCalendar

```
public GregorianCalendar(TimeZone zone,  
                        Locale aLocale)
```

Constructs a `GregorianCalendar` based on the current time in the given time zone with the given locale.

Parameters:

`zone` - the given time zone.

`aLocale` - the given locale.

GregorianCalendar

```
public GregorianCalendar(int year,  
                        int month,  
                        int date)
```

Constructs a `GregorianCalendar` with the given date set in the default time zone with the default locale.

Parameters:

`year` - the value used to set the YEAR time field in the calendar.

`month` - the value used to set the MONTH time field in the calendar. Month value is 0-based. e.g., 0 for January.

`date` - the value used to set the DATE time field in the calendar.

GregorianCalendar

```
public GregorianCalendar(int year,
                        int month,
                        int date,
                        int hour,
                        int minute)
```

Constructs a `GregorianCalendar` with the given date and time set for the default time zone with the default locale.

Parameters:

`year` - the value used to set the YEAR time field in the calendar.

`month` - the value used to set the MONTH time field in the calendar. Month value is 0-based. e.g., 0 for January.

`date` - the value used to set the DATE time field in the calendar.

`hour` - the value used to set the HOUR_OF_DAY time field in the calendar.

`minute` - the value used to set the MINUTE time field in the calendar.

GregorianCalendar

```
public GregorianCalendar(int year,
                        int month,
                        int date,
                        int hour,
                        int minute,
                        int second)
```

Constructs a `GregorianCalendar` with the given date and time set for the default time zone with the default locale.

Parameters:

`year` - the value used to set the YEAR time field in the calendar.

`month` - the value used to set the MONTH time field in the calendar. Month value is 0-based. e.g., 0 for January.

`date` - the value used to set the DATE time field in the calendar.

`hour` - the value used to set the HOUR_OF_DAY time field in the calendar.

`minute` - the value used to set the MINUTE time field in the calendar.

`second` - the value used to set the SECOND time field in the calendar.

Method Detail

setGregorianChange

```
public void setGregorianChange(Date date)
```

Sets the GregorianCalendar change date. This is the point when the switch from Julian dates to Gregorian dates occurred. Default is October 15, 1582. Previous to this, dates will be in the Julian calendar.

To obtain a pure Julian calendar, set the change date to `Date(Long.MAX_VALUE)`. To obtain a pure Gregorian calendar, set the change date to `Date(Long.MIN_VALUE)`.

Parameters:

date - the given Gregorian cutover date.

getGregorianChange

```
public final Date getGregorianChange()
```

Gets the Gregorian Calendar change date. This is the point when the switch from Julian dates to Gregorian dates occurred. Default is October 15, 1582. Previous to this, dates will be in the Julian calendar.

Returns:

the Gregorian cutover date for this calendar.

isLeapYear

```
public boolean isLeapYear(int year)
```

Determines if the given year is a leap year. Returns true if the given year is a leap year.

Parameters:

year - the given year.

Returns:

true if the given year is a leap year; false otherwise.

equals


```
public boolean equals(Object obj)
```

Compares this GregorianCalendar to an object reference.

Overrides:

[equals](#) in class [Calendar](#)

Parameters:

obj - the object reference with which to compare

Returns:

true if this object is equal to obj; false otherwise

hashCode

```
public int hashCode()
```

Override hashCode. Generates the hash code for the GregorianCalendar object

Overrides:

[hashCode](#) in class [Calendar](#)

Returns:

a hash code value for this object.

add

```
public void add(int field,
               int amount)
```

Adds the specified (signed) amount of time to the given time field, based on the calendar's rules.

Add rule 1. The value of `field` after the call minus the value of `field` before the call is `amount`, modulo any overflow that has occurred in `field`. Overflow occurs when a field value exceeds its range and, as a result, the next larger field is incremented or decremented and the field value is adjusted back into its range.

Add rule 2. If a smaller field is expected to be invariant, but it is impossible for it to be equal to its prior value because of changes in its minimum or maximum after `field` is changed, then its value is adjusted to be as close as possible to its expected value. A smaller field represents a smaller unit of time. `HOUR` is a smaller field than `DAY_OF_MONTH`. No

adjustment is made to smaller fields that are not expected to be invariant. The calendar system determines what fields are expected to be invariant.

Specified by:

[add](#) in class [Calendar](#)

Parameters:

`field` - the time field.

`amount` - the amount of date or time to be added to the field.

Throws:

[IllegalArgumentException](#) - if an unknown field is given.

roll

```
public void roll(int field,
                  boolean up)
```

Adds or subtracts (up/down) a single unit of time on the given time field without changing larger fields.

Example: Consider a `GregorianCalendar` originally set to December 31, 1999. Calling `roll(Calendar.MONTH, true)` sets the calendar to January 31, 1999. The `Year` field is unchanged because it is a larger field than `MONTH`.

Specified by:

[roll](#) in class [Calendar](#)

Parameters:

`up` - indicates if the value of the specified time field is to be rolled up or rolled down. Use true if rolling up, false otherwise.

`field` - the time field.

Throws:

[IllegalArgumentException](#) - if an unknown field value is given.

See Also:

[add\(int, int\)](#), [Calendar.set\(int, int\)](#)

roll

```
public void roll(int field,
                  int amount)
```

Add to field a signed amount without changing larger fields. A negative roll amount means to

subtract from field without changing larger fields.

Example: Consider a `GregorianCalendar` originally set to August 31, 1999. Calling `roll(Calendar.MONTH, 8)` sets the calendar to April 30, **1999**. Using a `GregorianCalendar`, the `DAY_OF_MONTH` field cannot be 31 in the month April. `DAY_OF_MONTH` is set to the closest possible value, 30. The `YEAR` field maintains the value of 1999 because it is a larger field than `MONTH`.

Example: Consider a `GregorianCalendar` originally set to Sunday June 6, 1999. Calling `roll(Calendar.WEEK_OF_MONTH, -1)` sets the calendar to Tuesday June 1, 1999, whereas calling `add(Calendar.WEEK_OF_MONTH, -1)` sets the calendar to Sunday May 30, 1999. This is because the roll rule imposes an additional constraint: The `MONTH` must not change when the `WEEK_OF_MONTH` is rolled. Taken together with add rule 1, the resultant date must be between Tuesday June 1 and Saturday June 5. According to add rule 2, the `DAY_OF_WEEK`, an invariant when changing the `WEEK_OF_MONTH`, is set to Tuesday, the closest possible value to Sunday (where Sunday is the first day of the week).

Overrides:

[roll](#) in class [Calendar](#)

Parameters:

`field` - the time field.

`amount` - the signed amount to add to `field`.

Since:

1.2

See Also:

[add\(int, int\)](#), [Calendar.set\(int, int\)](#)

getMinimum

```
public int getMinimum(int field)
```

Returns minimum value for the given field. e.g. for Gregorian `DAY_OF_MONTH`, 1 Please see `Calendar.getMinimum` for descriptions on parameters and the return value.

Specified by:

[getMinimum](#) in class [Calendar](#)

Parameters:

`field` - the given time field.

Returns:

the minimum value for the given time field.

getMaximum

```
public int getMaximum(int field)
```

Returns maximum value for the given field. e.g. for Gregorian DAY_OF_MONTH, 31 Please see Calendar.getMaximum for descriptions on parameters and the return value.

Specified by:

[getMaximum](#) in class [Calendar](#)

Parameters:

`field` - the given time field.

Returns:

the maximum value for the given time field.

getGreatestMinimum

```
public int getGreatestMinimum(int field)
```

Returns highest minimum value for the given field if varies. Otherwise same as getMinimum (). For Gregorian, no difference. Please see Calendar.getGreatestMinimum for descriptions on parameters and the return value.

Specified by:

[getGreatestMinimum](#) in class [Calendar](#)

Parameters:

`field` - the given time field.

Returns:

the highest minimum value for the given time field.

getLeastMaximum

```
public int getLeastMaximum(int field)
```

Returns lowest maximum value for the given field if varies. Otherwise same as getMaximum (). For Gregorian DAY_OF_MONTH, 28 Please see Calendar.getLeastMaximum for descriptions on parameters and the return value.

Specified by:

[getLeastMaximum](#) in class [Calendar](#)

Parameters:

`field` - the given time field.

Returns:

the lowest maximum value for the given time field.

getActualMinimum

```
public int getActualMinimum(int field)
```

Return the minimum value that this field could have, given the current date. For the Gregorian calendar, this is the same as `getMinimum()` and `getGreatestMinimum()`.

Overrides:

[getActualMinimum](#) in class [Calendar](#)

Parameters:

`field` - the field to determine the minimum of

Returns:

the minimum of the given field for the current date of this Calendar

Since:

1.2

getActualMaximum

```
public int getActualMaximum(int field)
```

Return the maximum value that this field could have, given the current date. For example, with the date "Feb 3, 1997" and the `DAY_OF_MONTH` field, the actual maximum would be 28; for "Feb 3, 1996" it s 29. Similarly for a Hebrew calendar, for some years the actual maximum for `MONTH` is 12, and for others 13.

Overrides:

[getActualMaximum](#) in class [Calendar](#)

Parameters:

`field` - the field to determine the maximum of

Returns:

the maximum of the given field for the current date of this Calendar

Since:

1.2

computeFields

protected void **computeFields**()

Converts UTC as milliseconds to time field values. The time is *not* recomputed first; to recompute the time, then the fields, call the `complete` method.

Specified by:

[computeFields](#) in class [Calendar](#)

See Also:

[Calendar.complete\(\)](#)

computeTime

protected void **computeTime**()

Overrides Calendar Converts time field values to UTC as milliseconds.

Specified by:

[computeTime](#) in class [Calendar](#)

Throws:

[IllegalArgumentException](#) - if any fields are invalid.

[Overview](#) [Package](#) [Class](#) [Use Tree](#) [Deprecated](#) [Index](#) [Help](#)

*Java™ 2 Platform
Std. Ed. v1.4.2*

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Submit a bug or feature](#)

For further API reference and developer documentation, see [Java 2 SDK SE Developer Documentation](#). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 1994-2004 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](#). Also see the [documentation redistribution policy](#).

A Summary of the International Standard Date and Time Notation

by Markus Kuhn

International Standard ISO 8601 specifies numeric representations of date and time. This standard notation helps to avoid confusion in international communication caused by the many different national notations and increases the portability of computer user interfaces. In addition, these formats have several important advantages for computer usage compared to other traditional date and time notations. The time notation described here is already the de-facto standard in almost all countries and the date notation is becoming increasingly popular.

Especially authors of Web pages and software engineers who design user interfaces, file formats, and communication protocols should be familiar with ISO 8601.

Contents: [Date](#), [Time of Day](#), [Time Zone](#).

Date

The international standard date notation is

YYYY-MM-DD

where YYYY is the year in the usual Gregorian calendar, MM is the month of the year between 01 (January) and 12 (December), and DD is the day of the month between 01 and 31.

For example, the fourth day of February in the year 1995 is written in the standard notation as

1995-02-04

Other commonly used notations are e.g. 2/4/95, 4/2/95, 95/2/4, 4.2.1995, 04-FEB-1995, 4-February-1995, and many more. Especially the first two examples are dangerous, because as both are used quite often in the U.S. and in Great Britain and both can not be distinguished, it is unclear whether 2/4/95 means 1995-04-02 or 1995-02-04. The date notation 2/4/5 has at least six reasonable interpretations (assuming that only the twentieth and twenty-first century are reasonable candidates in our life time).

Advantages of the ISO 8601 standard date notation compared to other commonly used variants:

- easily readable and writeable by software (no 'JAN', 'FEB', ... table necessary)
- easily comparable and sortable with a trivial string comparison

- language independent
- can not be confused with other popular date notations
- consistency with the common 24h time notation system, where the larger units (hours) are also written in front of the smaller ones (minutes and seconds)
- strings containing a date followed by a time are also easily comparable and sortable (e.g. write "1995-02-04 22:45:00")
- the notation is short and has constant length, which makes both keyboard data entry and table layout easier
- identical to the Chinese date notation, so the largest cultural group (>25%) on this planet is already familiar with it :-)
- date notations with the order "year, month, day" are in addition already widely used e.g. in Japan, Korea, Hungary, Sweden, Finland, Denmark, and a few other countries and people in the U.S. are already used to at least the "month, day" order
- a 4-digit year representation avoids overflow problems after 2099-12-31

As dates will look a little bit strange anyway starting with 2000-01-01 (e.g. like 1/1/0), it has been suggested that the year 2000 is an excellent opportunity to change to the standard date notation.

ISO 8601 is only specifying numeric notations and does not cover dates and times where words are used in the representation. It is not intended as a replacement for language-dependent worded date notations such as "24. Dezember 2001" (German) or "February 4, 1995" (US English). ISO 8601 should however be used to replace notations such as "2/4/95" and "9.30 p.m.".

Apart from the recommended primary standard notation **YYYY-MM-DD**, ISO 8601 also specifies a number of alternative formats for use in applications with special requirements. All of these alternatives can easily and automatically be distinguished from each other:

The hyphens can be omitted if compactness of the representation is more important than human readability, for example as in

19950204

For situations where information about the century is really not required, a 2-digit year representation is available:

95-02-04 or **950204**

If only the month or even only the year is of interest:

1995-02 or **1995**

In commercial and industrial applications (delivery times, production plans, etc.), especially in Europe, it is often required to refer to a week of a year. Week 01 of a year is per definition the first week that has the Thursday in this year, which is equivalent to the week that contains the fourth day of January. In other words, the first week of a new year is the week that has the majority of its days in

the new year. Week 01 might also contain days from the previous year and the week before week 01 of a year is the last week (52 or 53) of the previous year even if it contains days from the new year. A week starts with Monday (day 1) and ends with Sunday (day 7). For example, the first week of the year 1997 lasts from 1996-12-30 to 1997-01-05 and can be written in standard notation as

1997-W01 or 1997W01

The week notation can also be extended by a number indicating the day of the week. For example, the day 1996-12-31, which is the Tuesday (day 2) of the first week of 1997, can also be written as

1997-W01-2 or 1997W012

for applications like industrial planning where many things like shift rotations are organized per week and knowing the week number and the day of the week is more handy than knowing the day of the month.

An abbreviated version of the year and week number like

95W05

is sometimes useful as a compact code printed on a product that indicates when it has been manufactured.

The ISO standard avoids explicitly stating the possible range of week numbers, but this can easily be deduced from the definition:

Theorem: Possible ISO week numbers are in the range 01 to 53. A year always has a week 52. (There is one historic exception: the year in which the Gregorian calendar was introduced had less than 365 days and less than 52 weeks.)

Proof: Per definition, the first week of a year is W01 and consequently days before week W01 belong to the previous year and so there is no week with lower numbers. Considering the highest possible week number, the worst case is a leap year like 1976 that starts with a Thursday, because this keeps the highest possible number of days of W01 in the previous year, i.e. 3 days. In this case, the Sunday of W52 of the worst case year is day number $4 + 51 \cdot 7 = 361$ and $361 - 366 = 5$ days of W53 belong still to this year, which guarantees that in the worst case year day 4 (Thursday) of W53 is not yet in the next year, so a week number 53 is possible. For example, the 53 weeks of the worst case year 1976 started with 1975-12-29 = 1976-W01-1 and ended with 1977-01-02 = 1976-W53-7. On the other hand, considering the lowest number of the last week of a year, the worst case is a non-leap year like 1999 that starts with a Friday, which ensures that the first three days of the year belong to the last week of the previous year. In this case, the Sunday of week 52 would be day number $3 + 52 \cdot 7 = 367$, i.e. only the last $367 - 365 = 2$ days of the W52 reach into the next year and consequently, even a worst case year like 1999 has a week W52 including the days 1999-12-27 to 2000-01-02. q.e.d.

[The new 1999 version of the C programming language standard (ISO 9899) added in the `strftime` () function means to generate the ISO 8601 week notation. The author of this text developed a further proposal for a [modernised clock and calendar API](#) for C, which provides full proper treatment of leap seconds and timezones and fixes numerous other problems in the current C timing library functions. It also serves as a model for those who want to design clock library functions for other programming languages.]

Both day and year are useful units of structuring time, because the position of the sun on the sky, which influences our lives, is described by them. However the 12 months of a year are of some obscure mystic origin and have no real purpose today except that people are used to having them (they do not even describe the current position of the moon). In some applications, a date notation is preferred that uses only the year and the day of the year between 001 and 365 (366 in leap years). The standard notation for this variant representing the day 1995-02-04 (that is day 035 of the year 1995) is

1995-035 or 1995035

Leap years are years with an additional day YYYY-02-29, where the year number is a multiple of four with the following exception: If a year is a multiple of 100, then it is only a leap year if it is also a multiple of 400. For example, 1900 was not a leap year, but 2000 is one.

Time of Day

The international standard notation for the time of day is

hh:mm:ss

where hh is the number of complete hours that have passed since midnight (00-24), mm is the number of complete minutes that have passed since the start of the hour (00-59), and ss is the number of complete seconds since the start of the minute (00-60). If the hour value is 24, then the minute and second values must be zero. [The value 60 for ss might sometimes be needed during an inserted [leap second](#) in an atomic time scale like Coordinated Universal Time (UTC). A single leap second 23:59:60 is inserted into the UTC time scale every few years as announced by the [International Earth Rotation Service](#) in Paris to keep UTC from wandering away more than 0.9 s from the less constant astronomical time scale UT1 that is defined by the actual rotation of the earth.]

An example time is

23:59:59

which represents the time one second before midnight.

As with the date notation, the separating colons can also be omitted as in

235959

and the precision can be reduced by omitting the seconds or both the seconds and minutes as in

23:59, 2359, or 23

It is also possible to add fractions of a second after a decimal dot or comma, for instance the time 5.8 ms before midnight can be written as

23:59:59.9942 or 235959.9942

As every day both starts and ends with midnight, the two notations **00:00** and **24:00** are available to distinguish the two midnights that can be associated with one date. This means that the following two notations refer to exactly the same point in time:

1995-02-04 24:00 = 1995-02-05 00:00

In case an unambiguous representation of time is required, 00:00 is usually the preferred notation for midnight and not 24:00. Digital clocks display 00:00 and not 24:00.

ISO 8601 does not specify, whether its notations specify a point in time or a time period. This means for example that ISO 8601 does not define whether 09:00 refers to the exact end of the ninth hour of the day or the period from 09:00 to 09:01 or anything else. The users of the standard must somehow agree on the exact interpretation of the time notation if this should be of any concern.

If a date and a time are displayed on the same line, then always write the date in front of the time. If a date and a time value are stored together in a single data field, then ISO 8601 suggests that they should be separated by a latin capital letter T, as in **19951231T235959**.

A remark for readers from the U.S.:

The 24h time notation specified here has already been the de-facto standard all over the world in written language for decades. The only exception are a few English speaking countries, where still notations with hours between 1 and 12 and additions like "a.m." and "p.m." are in wide use. The common 24h international standard notation is widely used now even in England (e.g. at airports, cinemas, bus/train timetables, etc.). Most other languages don't even have abbreviations like "a.m." and "p.m." and the 12h notation is certainly hardly ever used on Continental Europe to write or display a time. Even in the U.S., the military and computer programmers have been using the 24h notation for a long time.

The old English 12h notation has many disadvantages like:

- It is longer than the normal 24h notation.

- It takes somewhat more time for humans to compare two times in 12h notation.
- It is not clear, how 00:00, 12:00 and 24:00 are represented. Even encyclopedias and style manuals contain contradicting descriptions and a common quick fix seems to be to avoid "12:00 a.m./p.m." altogether and write "noon", "midnight", or "12:01 a.m./p.m." instead, although the word "midnight" still does not distinguish between 00:00 and 24:00 (midnight at the start or end of a given day).
- It makes people often believe that the next day starts at the overflow from "12:59 a.m." to "1:00 a.m.", which is a common problem not only when people try to program the timer of VCRs shortly after midnight.
- It is not easily comparable with a string compare operation.
- It is not immediately clear for the unaware, whether the time between "12:00 a.m./p.m." and "1:00 a.m./p.m." starts at 00:00 or at 12:00, i.e. the English 12h notation is more difficult to understand.

Please consider the 12h time to be a relic from the dark ages when Roman numerals were used, the number zero had not yet been invented and analog clocks were the only known form of displaying a time. Please avoid using it today, especially in technical applications! Even in the U.S., the widely respected *Chicago Manual of Style* now recommends using the international standard time notation in publications.

A remark for readers from German speaking countries:

The German standard DIN 5008, which specifies typographical rules for German texts written on typewriters, was updated in 1996-05. The old German numeric date notations DD.MM.YYYY and DD.MM.YY have been replaced by the ISO date notations YYYY-MM-DD and YY-MM-DD. Similarly, the old German time notations hh.mm and hh.mm.ss have been replaced by the ISO notations hh:mm and hh:mm:ss. Those new notations are now also mentioned in the latest edition of the *Duden*. The German alphanumeric date notation continues to be for example "3. August 1994" or "3. Aug. 1994". The corresponding Austrian standard has already used the ISO 8601 date and time notations before.

ISO 8601 has been adopted as European Standard EN 28601 and is therefore now a valid standard in all EU countries and all conflicting national standards have been changed accordingly.

Time Zone

Without any further additions, a date and time as written above is assumed to be in some local time zone. In order to indicate that a time is measured in [Universal Time \(UTC\)](#), you can append a capital letter **Z** to a time as in

23:59:59Z or **2359Z**

[The Z stands for the "zero meridian", which goes through Greenwich in London, and it is also commonly used in radio communication where it is pronounced "Zulu" (the word for Z in the international radio alphabet). [Universal Time](#) (sometimes also called "Zulu Time") was called Greenwich Mean Time (GMT) before 1972, however this term should no longer be used. Since the introduction of an international atomic time scale, almost all existing civil time zones are now related to UTC, which is slightly different from the old and now unused GMT.]

The strings

+hh:mm, +hhmm, or +hh

can be added to the time to indicate that the used local time zone is hh hours and mm minutes ahead of UTC. For time zones west of the zero meridian, which are behind UTC, the notation

-hh:mm, -hhmm, or -hh

is used instead. For example, Central European Time (CET) is +0100 and U.S./Canadian Eastern Standard Time (EST) is -0500. The following strings all indicate the same point of time:

12:00Z = 13:00+01:00 = 0700-0500

There exists no international standard that specifies abbreviations for civil time zones like CET, EST, etc. and sometimes the same abbreviation is even used for two very different time zones. In addition, politicians enjoy modifying the rules for civil time zones, especially for daylight saving times, every few years, so the only really reliable way of describing a local time zone is to specify numerically the difference of local time to UTC. Better use directly UTC as your only time zone where this is possible and then you do not have to worry about time zones and daylight saving time changes at all.

More Information about Time Zones

Arthur David Olson and others maintain a [database of all current and many historic time zone changes and daylight saving time algorithms](#). It is available via ftp from [elsie.nci.nih.gov](ftp://elsie.nci.nih.gov) in the `tzcode*` and `tzdata*` files. Most Unix time zone handling implementations are based on this package. If you want to join the `tz` mailing list, which is dedicated to discussions about time zones and this software, please send a request for subscription to `tz-request` at elsie.nci.nih.gov. You can read previous discussion there in the [tz archive](#).

Other Links about Date, Time, and Calendars

Some other interesting sources of information about date and time on the Internet are for example the [Glossary of Frequency and Timing Terms](#) and the [FAQ](#) provided by [NIST](#), the [Yahoo Science: Measurements and Units:Time](#) link collection, the [U.S. Naval Observatory Server](#), the [International Earth Rotation Service \(IERS\)](#) (for time gurus only!), the [Network Time Protocol \(NTP\)](#), the time

and calendar section of the [USENET sci.astro FAQ](#), and the [Calendar FAQ](#).

This was a brief overview of the ISO 8601 standard, which covers only the most useful notations and includes some additional related information. The full standard defines in addition a number of more exotic notations including some for periods of time. The ISO 8601:2000 standard itself can only be ordered on paper or as a PDF file on CD-ROM either via ISO's web site [online](#) or from

[International Organization for Standardization](#)

Case postale 56
1, rue de Varembé
CH-1211 Genève 20
Switzerland

phone: +41 22 749 01 11
fax: +41 22 733 34 30
email: sales at isocs.iso.ch

A more detailed online summary of ISO 8601 than this one is the text *ISO 8601:1988 Date/Time Representations* available from <ftp.informatik.uni-erlangen.de/pub/doc/ISO/ISO8601.ps.Z> (PostScript, 16 kb, 5 pages) written by Gary Houston, now also available in [HTML](#). Ian Galpin (G1SMD) proposes to use ISO 8601 as a [Common Date-Time Standard for Amateur Radio](#). [Steve Adams](#) has written [another web page](#) about the ISO date format that is partially based on this text. Another summary of ISO 8601 is [Jukka Korpela's page](#) and there are further related pages listed in the [Open Directory](#).

The committee in charge of ISO 8601 is ISO TC 154 and the editor of the second edition ISO 8601:2000 was Louis Visser.

I wish to thank [Edward M. Reingold](#) for developing the fine GNU Emacs calendar functions, as well as Rich Wales, Mark Brader, Paul Eggert, and others in the [comp.std.internat](#), [comp.protocols.time.ntp](#), and [sci.astro](#) USENET discussion groups for valuable comments about this text. Further comments and hyperlinks to this page are very welcome.

Some journalists recently got interested in the international date and time format and reported about it. Examples include:

- An article by Jon G. Auerbach in the 1999-06-01 issue of the Wall Street Journal, page A1.

If you are a journalist and need information on this or related topics, please feel free to contact me.

You might also be interested in the [International Standard Paper Sizes](#) Web page.

[Markus Kuhn](#)

created 1995 -- last modified 2001-11-10 -- <http://www.cl.cam.ac.uk/~mgk25/iso-time.html>

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import javax.ejb.RemoveException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;

public class CallPersonBean {
    public static void main(String[] args) {
        try {
            Context initial = new InitialContext();
            Object objRef = initial.lookup("personBean");

            PersonHome home =
                (PersonHome) PortableRemoteObject.narrow(
                    objRef,
                    PersonHome.class);
            Person p1 = home.create(1911, 1, 1, "Alice", "streetA");
            Person p2 = home.create(1922, 2, 2, "Bob", "streetB");
            System.out.println("Alice's Age: " + p1.getAge());

            System.out.println("Alice's primary key: " + p1.getPrimaryKey());
            p1.remove();

            Person p3 = home.findByPrimaryKey("Bob");
            System.out.println(
                "Bob's modified street (before modification): "
                + p3.getStreet());
            p3.setStreet("streetC");
            System.out.println(
                "Bob's modified street (after modification): "
                + p3.getStreet());
            System.out.println("also the other reference: " + p2.getStreet());

            System.out.println("Are both references the same Java object: " + (p2==p3));
            System.out.println("Are both references the same EJBObject (calls isIdentical): " + p2.
isIdentical(p3));

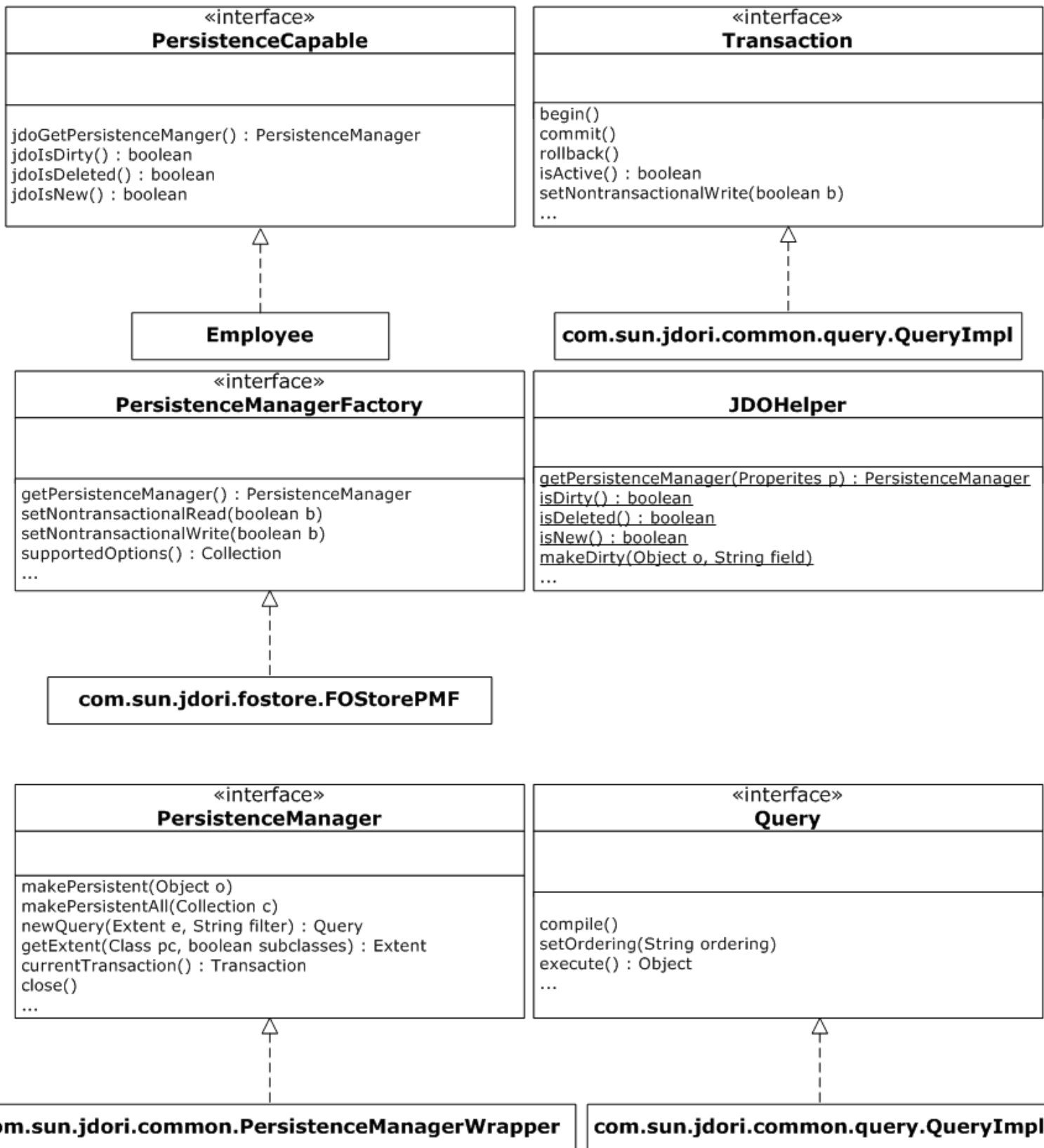
        } catch (NamingException ne) {
            ne.printStackTrace();
        } catch (RemoteException re) {
            re.printStackTrace();
        } catch (CreateException ce) {
```



```
        ce.printStackTrace();  
    } catch (RemoveException e) {  
        e.printStackTrace();  
    } catch (FinderException e) {  
        e.printStackTrace();  
    }  
}
```

```
}
```

```
javax.jdo.PersistenceManagerFactoryClass=com.sun.jdori.fostore.FOStorePMF
javax.jdo.option.ConnectionURL=fostore:jdoriDB
javax.jdo.option.ConnectionUserName=mario
javax.jdo.option.ConnectionPassword=thePassword
```



```
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.Transaction;

public class JDOCreateDB {
    public static void main(String args[]) {
        Properties props = new Properties();
        try {
            InputStream is = ClassLoader.getResourceAsStream("jdo.properties");
            props.load(is);
            props.put("com.sun.jdori.option.ConnectionCreate","true");
        } catch (IOException ioe) {
            System.out.println("Error loading properties");
            System.exit(1);
        }
        PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(props);
        PersistenceManager pm = pmf.getPersistenceManager();

        Transaction tx = pm.currentTransaction();
        tx.begin();
        tx.commit();
        pm.close();
    }
}
```

```
package de.jeckle.jdotest;

import java.util.HashSet;
import java.util.Iterator;

public class Employee {
    private String name;
    private String department;
    private HashSet projects = new HashSet();

    public String getName() {
        return name;
    }
    public String getDepartment() {
        return department;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setDepartment(String department) {
        this.department = department;
    }
    public void addProject(String project) {
        projects.add(project);
    }

    public String toString() {
        String result="Employee named "+name+" works in department "+department;
        result+="\nworks in: ";
        Iterator i = projects.iterator();
        while (i.hasNext()) {
            result+=(String) i.next();
            result+=", ";
        }
        return (result);
    }
}
```

▲ Vorlesung Java

▼ 1 Einführung in Java

▼ 1.1 Was ist Java?

▼ 1.2 Entstehungsgeschichte

▼ 1.3 Die Java-Plattform

▼ 1.4 Vom Quellcode zum lauffähigen Programm

▼ 2 Syntax und Semantik der Programmiersprache Java

▼ 2.1 C, C++, C# und Java

▼ 2.2 Grundstrukturen

▼ 2.2.1 Programmaufbau

▼ 2.2.2 Einfache Datentypen

▼ 2.2.3 Operatoren

▼ 2.3 Kontrollstrukturen

▼ 2.3.1 Selektion und Mehrfachselektion -- das if und case-Statement

▼ 2.3.2 Iteration -- for-, do-while-Schleifen

▼ 2.3.3 Ausnahmen und ihre Behandlung -- Exception Handling

▼ 2.3.4 Zwangsbedingungen

▼ 2.4 Von komplexen zu objektorientierten Datenstrukturen

▼ 2.4.1 Arrays

▼ 2.4.2 Klassen

▼ 2.4.3 Attribute

▼ 2.4.4 Operationen und Methoden

▼ 2.4.5 Aufzählungstypen

▼ 2.4.6 Wrapper-Typen

▼ 2.4.7 Boxing / Unboxing

▼ 2.4.8 Vererbung

▼ 2.4.9 Schnittstellen

▼ 2.4.10 Pakete

▼ 3 Die Java-Plattform

▼ 3.1 Die Laufzeitumgebung

▼ 3.1.1 Garbage Collection

▼ 3.1.2 Virtuelle Maschine

▼ 3.2 Die Java API und weiterführende Themen

▼ 3.2.1 Ein- / Ausgabe -- Streams

▼ 3.2.2 Threads und Nebenläufigkeit

▼ 3.2.3 Applets

▼ 3.2.4 Collection API

▼ 3.2.5 Generics

▼ 3.2.6 Reflection API

▼ 3.2.7 Abstract Windowing Toolkit (AWT)

▼ 3.2.8 Swing

▲ 1 Einführung in Java

▲ 1.1 Was ist Java?

- Laut *Brockhaus*:
 - von staatseigenen indones[ischen] Plantagen auf Java stammender Blatt-Tabak, der je nach Qualität als Deckblatt, Umblatt oder Einlage bei der Zigarrenherstellung verwendet wird.
Aus: Brockhaus --- Die Enzyklopädie, Bd. 11, 20. Aufl., 1997, p. 149.
 - **Java, Jawa** früher **Djava** [...], kleinste, aber bedeutendste der Großen Sundainseln, Indonesien, 1 060 km lang, 55-195 km breit, 118 000 km², (1990) 107,58 Mio. Ew.; verwaltungsmäßig (einschließlich der Insel Madura 132 186 km²) in drei Prov., die Sonderregion Yogyakarta und den hauptstadtsonderbezirk Jakarta gegliedert. Die Insel ist überwiegend gebirgig. Tätige und erloschene Einzelvulkane und Vulkangruppen bilden die zentrale Längsachse der Insel (mehrere Gipfel über 3 000m ü. M., Semeru 3 676m ü.M.). Beiderseits der Vulkanreihe (17 tätige Vulkane) gliedern sich Berg- und Hügelländer (aus tertiären Kalken und Mergeln) an. Im N[orden] sind dem Bergland z.T. versumpfte Schwemmlandebenen vorgelagert.
Aus: Brockhaus --- Die Enzyklopädie, Bd. 11, 20. Aufl., 1997, p. 149.
 - *Informatik*: objektorientierte, in starkem Maße an C++ angelehnte betriebssystem- und plattformunabhängige Programmiersprache, die 1995 von der Firma "Sun Microsystems" (USA) entwickelt wurde. J[ava] ist eine Sprache für in Netzen verteilbare Anwendungen, bei denen nicht nur Datensätze, sondern auch Programmteile (so genannte *Applets*;) in den Rechner geladen und ausgeführt werden können. Damit wird die in der Informatik, speziell im ->Internet, äußerst wichtige Forderung nach Systemunabhängigkeit erfüllt. J[ava]-Programme können auf beliebigen Rechnern ausgeführt werden, die über einen javafähigen WWW-Browser (->World Wide Web) verfügen; zusätzlich können wie --- wie herkömml[iche] Programme --- auch ohne Browser ausgeführt werden, sofern das J[ava]-Laufzeitsystem direkt im Betriebssystem implementiert ist.
Aus: Brockhaus --- Die Enzyklopädie, Bd. 11, 20. Aufl., 1997, p. 149.
- Umgangssprachlicher Ausdruck für starken, heißen Kaffee in den USA
- In einem Satz (nach der Definition von [Sun Microsystems](#)): eine einfache und kleine, objektorientierte, dezentrale, interpretierte, stabil laufende, architekturneutrale, portierbare und dynamische Sprache, die Hochgeschwindigkeitsanwendungen und Multithreading unterstützt.
Im einzelnen bedeutet das:
 - Java ist einfach; die Sprachkonzepte (Syntax) sind überschaubar. Im wesentlichen Stellen sie eine vereinfachte Untermenge der Programmiersprache C++ dar.
Wesentliche Unterschiede: stärker objektorientiert (d.h. keine *structs* mehr möglich, jedoch Beibehaltung des Konzepts der primitiven Werttypen), keine expliziten Zeiger (daher auch keine Zeigerarithmetik mehr möglich), keine Operatorüberladung, kein Präprozessor, keine Headerdateien (dafür Paketkonzept), automatische Speicherbereinigung durch Garbage Collection.
Klein zielt sowohl auf den Umfang der entstehenden ausführbaren Dateien, als auch die Größe der Java-Laufzeitumgebung ab. So werden Importdateien in Java nicht statisch inkludiert, sondern nur extern referenziert, wodurch die Größe teilweise signifikant verringert werden kann. Beispielsweise mißt das *HelloWorld*-Beispiel als Java-Bytecode 426 Byte, wohingegen das C-Pendant ([HelloWorld.c](#)) (gelinkt als command line executable für die Windows32-Plattform als [HelloWorld.exe](#)) über 34.000 Byte, bzw. unter Nutzung der Cygwin-Bibliothek mehr als 18.000 Byte, benötigt.
 - Eine einzige Basisklasse *Object* als Wurzel der gesamten Klassenhierarchie. Keine Mehrfachvererbung (aber Nachbildung mit *Interfaces* möglich, wodurch die wesentlichen Nachteile eliminiert werden, jedoch die Vorteile gewahrt bleiben).
Mit Java-Beans steht ein eigenes Komponentenmodell zur Verfügung.
 - Java erfüllt ein wesentliches Charakteristikum von Client/Server-Anwendungen, die Fähigkeit, Informationen und die Berechnungslast der Daten zu verteilen. Der Begriff *dezentral* beschreibt dabei die Beziehung zwischen Systemobjekten, gleichgültig ob diese Objekte auf lokalen oder entfernten Systemen abgelegt sind.
Am bekanntesten dürften die Java *Applets* sein, welche die Verteilung ganzer Applikationen über das World-Wide-Web (Abk. WWW).
 - Mit Java erstellte Applikationen sind auf jeder Plattform (Hardware, Betriebssystem) ausführbar, für die eine Java Virtual Machine (Abk. JVM) existiert. Um dies zu erreichen wird durch den Java-Compiler kein plattformspezifischer Objektcode generiert, sondern eine *Zwischenrepräsentation* --- der *Byte-Code*. Dieser wird zur Ausführungszeit (auch: Laufzeit) durch das *Laufzeitsystem* --- die JVM --- *interpretiert* und für den Anwender transparent in maschinenspezifische Instruktionsfolgen umgesetzt. Eine JVM kann auch im *Web Browser* implementiert sein (z.B. [Netscape Navigator](#), [Microsoft Internet-Explorer](#)).
 - Java ist *stabil* im Sinne von *zuverlässig*. Dies bedeutet in der Praxis die Verringerung der Absturzwahrscheinlichkeit (gleichbedeutend mit undefiniertem, unvorherseh- und -sagbarem

Verhalten) eines mit der Sprache geschriebenen Programms.

Java ist (wie Pascal oder Oberon) eine streng typisierte Sprache, das bedeutet Typfehler können bereits zur Compilerzeit erkannt werden.

Laufzeitfehler können in Form von expliziten Ausnahmeroutinen behandelt werden (Exception Handling).

Durch den Verzicht auf explizite Zeiger wurde eine häufige Quelle für Laufzeit- und Speicherfehler eliminiert.

- Die Neutralität hinsichtlich der zugrundeliegenden Zielarchitektur (Hardware, Betriebssystem) wurde bereits bei der JVM angesprochen. Hier soll nochmals der Aspekt des architekturunabhängigen Bytecodes betont werden.
- Folgt (eigentlich) direkt aus der [Architekturneutralität](#); die Java-Laufzeitumgebung --- die virtuelle Maschine --- kann für beliebige Zielarchitekturen umgesetzt werden. Sofort danach sind alle Java-Applikationen, ohne Änderung, auf der neuen Plattform ablauffähig. Das selbe gilt für mit Java erstellte Oberflächen, die ebenfalls nicht auf eine Plattform (Betriebssystem oder Window Manager) festgelegt sind. Der --- bei nativ compilierenden Sprachen wie C/C++ notwendige --- Aufwand eine Applikation an eine neue Zielumgebung anzupassen (d.h. Portierung auf neue Hardware oder Betriebssystem (Zeigerlängen, Datentypgrößen, ...) entfällt durch die Zwischenstufe der JVM, bzw. wird in die Umsetzung dieser auf die neuen Gegebenheiten verlagert.
- Die Architektur von Java ermöglicht es, die Sprache jederzeit an weiterentwickelnde Umgebungen anzupassen.
- Es versteht sich von selbst, dass plattformunabhängig ausgelagter Bytecode der zur Laufzeit jedesmal interpretiert und in plattformspezifischen Maschinencode umgesetzt werden muss, Geschwindigkeitsnachteile gegenüber auf genau eine Plattform zugeschnittenen Applikationen aufweist. Jedoch haben Untersuchungen von Sun gezeigt, dass Java-Bytecode --- der in plattformspezifischen Maschinencode übersetzt wurde --- plattformspezifischem Code anderer Sprachen ebenbürtig ist. (Hierdurch geht jedoch der Aspekt der Plattformunabhängigkeit verloren!). Java ist jedoch reinen Scriptsprachen (wie Perl oder Python) an Ausführungsgeschwindigkeit signifikant überlegen. Im Allgemeinen ist ein Java-Programm ca. 15-25 mal langsamer als optimierter C-Code. (Durch *Just in Time Compilation* läßt sich dieses Manko jedoch zumindest abfedern. Mittlerweile existieren auch plattformabhängige Compiler für Java, die nativen Binärcode erzeugen. Dieser erreicht üblicherweise die Geschwindigkeit von C++-Applikationen.)
- *Multithreading* soll (zunächst) vereinfacht als die Fähigkeit eines Programms definiert werden mehrere Tätigkeiten zur selben Zeit ausführen zu können. Zur Synchronisation wird das Monitorkonzept nach Hoare direkt in der API realisiert.

▲ 1.2 Entstehungsgeschichte

- 1990: Auf Anraten des Programmierers Patrick Naughton wird bei [Sun Microsystems](#) eine Entwicklertruppe unter dem Namen *Green* ins Leben gerufen. Zentrale Mitarbeiter sind, neben Naughton, James Gosling und Mike Sheridan. Ziel des (anfänglich hochgeheimen) Projekts ist die Analyse des EDV-Marktes hinsichtlich verwertbarer Zukunftstrends. Die Männer ermitteln als zukünftig erfolgversprechende Einnahmequelle für SUN den Bereich der Consumerelektronik (d.h. Telephone, Videorecorder, Wasch- und Kaffeemaschinen --- kurz allerlei elektronische Alltagsgeräte). Ferner definieren sie Anforderungen an ein zukünftiges plattformunabhängiges (d.h. universell einsetzbares) Betriebssystem dieser Maschinen. Vor allem Fehlertoleranz sowie leichte Bedienbarkeit und besondere Stabilität sollten dieses gegenüber existierenden Betriebssystemen auszeichnen. Die Softwaresteuerung sollte in C++ realisiert werden.
- Frühjahr 1991: erster Prototyp eines solchen Systems
- August 1991: Unter Federführung von James Gosling wird nach einer Sprache zur Programmierung des neu zu entwickelnden Betriebssystems gesucht; leider erfolglos. Nicht das es im Sommer 1991 an möglichen Programmiersprachenaspiranten mangelte, sondern keiner der Kandidaten deckte das im Jahr zuvor formulierte Anforderungsspektrum zur Zufriedenheit ab. Folglich wurde eine neue Sprache --- unter dem Namen *Oak* (engl. Eiche) --- entwickelt.
- 1992: Green-Team präsentiert ersten Prototyp ([Trickfigur Duke](#)) Die unabhängige Firma *FirstPerson* wird gegründet.
- März 1993: Neuorientierung des Projekts und Konzentration der Aktivitäten auf den Bereich interaktiven Fernsehens.
- April 1993: Der von Marc Andressen entwickelte erste graphische WWW-Browser NCSA Mosaic wird vorgestellt.
- Sept. 1993: Arthur van Hoff stößt zur Mannschaft von First Person.
- bis 1994 ist es FirstPerson nicht gelungen Oak zu vermarkten --- die Firma wird eingestellt, und

das Produkt verschwindet in der Versenkung. Das Green-Team kehrt zu Sun zurück.

- 1994: Auf Initiative von William (Bill) Joy wird Oak für das Internet modifiziert (der mittlerweile legendäre *Mosaic*-Browser war ein Nebenprodukt der Arbeit des Green-Team).
- 1995: Oak wird im Internet zur Verfügung gestellt; allerdings nun unter dem neuen Namen *Java*. Vorstellung der β -Version auf Sun-World-Konferenz.
- 1996: Java Development Kit (Abk. JDK) v1.0
- 1997: JDK v1.1
- 1998: JDK v1.2
- 2000: JDK v1.3
- 2002: JDK: v1.4
- 2003: Erste Betaversion des JDK v1.5 (Codename *Tiger*)



Abbildung 1: Das alte und neue Javalogo sowie Duke das Maskottchen (click on image to enlarge!)

▲ 1.3 Die Java-Plattform

Prinzipiell kann, wie erwähnt, eine Java-Applikation auf jeder beliebigen Plattform zur Ausführung gebracht werden --- unabhängig davon welches Gerät die notwendige virtuelle Maschine implementiert.

So existieren Visionen und erste Umsetzungen von JVMs für Elektrogeräte wie Kaffeemaschinen, Kühlschränke, Radios etc.

Zur Erinnerung: die Intention der ursprünglichen Java-Entwicklung zielte nicht auf Desktop Rechner oder gar das Internet.

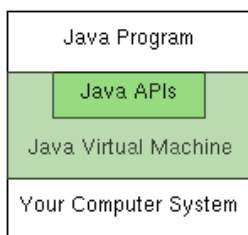


Abbildung 2: Aufbau der Java Plattform (click on image to enlarge!)

Die Abbildung schematisiert den Aufbau der Java-Plattform. Von der konkreten Plattform der physischen Hardware (Prozessor, etc. und des darauf ausgeführten Betriebssystems) wird durch die plattformspezifische virtuelle Maschine abstrahiert.

In diese eingebettet ist das *Java-Application Programmers Interface* (Abk. API) --- die sog. *Klassenbibliothek*. Sie ist vollständig in Java realisiert, damit ist sie bereits als plattformunabhängiges Java-Programm ausgelegt. Das API liefert die wesentlichen Grundprimitiven zur Erstellung leistungsfähiger Programme.

Auf dieser *virtuellen Plattform* läuft das (Benutzer) Java-Programm --- in Form des Bytecodes --- ab.

Hinweis: Beachten Sie die Unterscheidung zwischen Java-Plattform und Ausführungsplattform (=Betriebssystem und Hardware) einer Java-Applikation

Zur Applikationserstellung mit der Java-Plattform bietet SUN drei verschiedene Ausbaustufen (*Editionen*) an, die in der nachfolgenden Abbildung zusammengestellt sind.

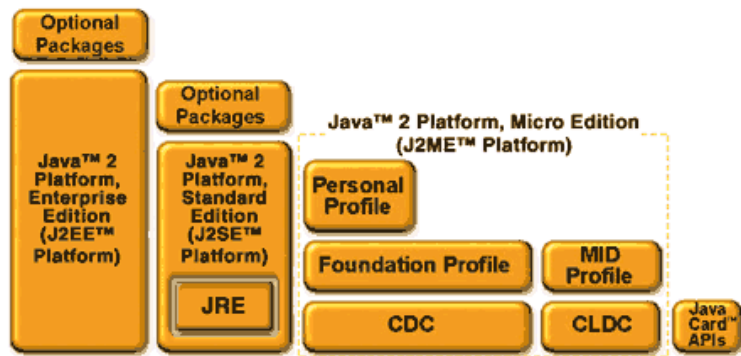


Abbildung 3: Java-Editionen (click on image to enlarge!)

Diese Editionen basieren alle auf demselben Java-Sprachkern, verfügen jedoch über verschiedene Standard-APIs, die auf die jeweils adressierte Problemstellung zugeschnitten sind. Im Einzelnen sind dies:

- **Java 2 Enterprise Edition:** Erstellung von Webapplikationen, die durch einen Application Server ausgeführt werden.
- **Java 2 Standard Edition:** Erstellung von Clientapplikationen zur Ausführung auf Desktopmaschinen.
- **Java 2 Micro Edition:** Erstellung von Applikationen für ressourcenbeschränkte Geräte wie Mobiltelefone, PDAs, etc.

Wir werden uns nachfolgend auf die Betrachtung des Java-Sprachkernes, sowie ausgewählter APIs beschränken, überwiegend in allen drei Editionen, jedoch mindestens in J2EE und J2SE, zur Verfügung stehen.

Das Sun Java Development Kit (JDK)

Java™ 2 Platform, Standard Edition v 1.4

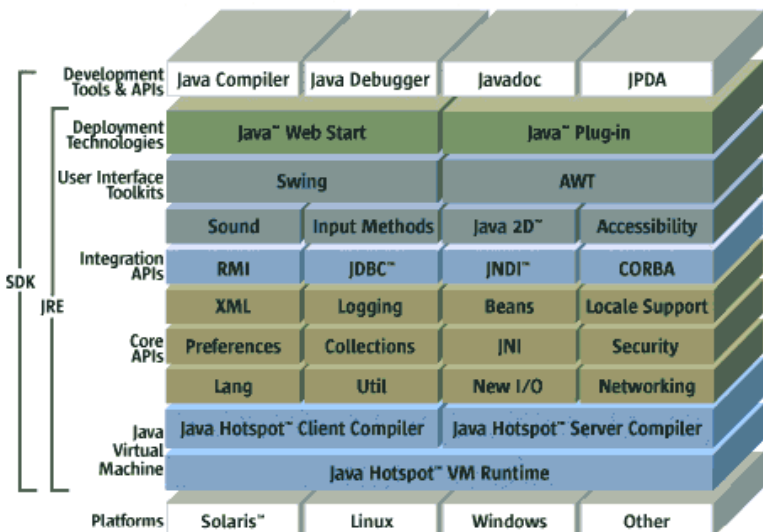


Abbildung 4: Architektur des Java SDK (JDK) von SUN (click on image to enlarge!)

Das JDK stellt eine Referenzimplementierung der kompletten Java-Plattform zur Verfügung. Seine Hauptbestandteile sind:

- **Java-Compiler:** javac
erzeugt Byte-Code
Minimalaufruf: javac example.java
Beispiel: javac [HelloWorld.java](#) erzeugt die Dateien [HelloWorld.class](#) und [SayHello.class](#)
- **Java-Interpreter:** java
Laufzeitumgebung zur Ausführung der Bytecode-Dateien
Minimalaufruf: java example
Wichtig: die Dateierweiterung .class darf *nicht* angegeben werden!
Beispiel: java HelloWorld führt die Applikation aus.
- **Applet-Viewer:** appletviewer
Dient zum Testen von Java-Applets ohne Web-Browser

Minimalaufruf: `appletviewer example.html`

Hinweis: Das Applet muß in einer Minimal-HTML-Seite referenziert werden, eine losgelöste Ausführung ist nicht möglich.

Beispiel (absolutes Minimalapplet ([HelloWorldApplet1.java](#)): `appletviewer HelloWorldApplet1.html`

- Dokumentationswerkzeug: `javadoc`
Erzeugt standardisierte HTML-Dokumentation aus Java-Quellcode und darin enthaltenen speziellen formalisierten Kommentaren.
Minimalaufruf: `javadoc example.java`
Beispiel: `javadoc -private -version -author -windowtitle "Beispiel einer JavaDoc generierten Dokumentation" Hello World.java`
Generiert verschiedenste Dokumentationsdateien im HTML- bzw. CSS-Format. Als Einstiegspunkt empfiehlt sich die Datei [index.html](#)
- Textmode-Debugger: `jdb`
Einfacher textorientierter Debugger
Minimalaufruf: `jdb example`
- Disassembler: `javap`
Rückübersetzer zur Wiedergewinnung von Java-Quellcode aus Bytecode-Dateien
Minimalaufruf: `javap example`

Die Rückübersetzung des Compilates [HelloWorld.class](#) mit `javap -c HelloWorld` liefert:

```
(1)Compiled from HelloWorld.java
(2)public class HelloWorld extends java.lang.Object {
(3)    public HelloWorld();
(4)    public static void main(java.lang.String[]);
(5)}
(6)
(7)Method HelloWorld()
(8)    0 aload_0
(9)    1 invokespecial #1 <Method java.lang.Object()>;
(10)   4 return
(11)
(12)Method void main(java.lang.String[])
(13)   0 getstatic #2 <Field java.io.PrintStream out>;
(14)   3 ldc #3 <String "Hello World!">;
(15)   5 invokevirtual #4 <Method void println(java.lang.String)>;
(16)   8 return
```

Beispiel 1: Decompilierung mit `javap`

Anmerkung: Zum Investitionsschutz kommerziell erstellter Java-Software sind Applikationen verfügbar, die den generierten Bytecode so nachbearbeiten, daß eine Decompilierung deutlich erschwert wird, oder die Ausgabe unbrauchbar wird. Die Ausführbarkeit des Bytecodes wird hierdurch nicht beeinträchtigt.

Zusätzlich ist ein eigenständiger Java-Interpreter, das sog. *Java Runtime Environment* (Abk. JRE) verfügbar. Sein Minimalaufruf lautet `jre example`. Diese Applikation ist nicht Bestandteil der Standard Java-Plattform (JDK v1.3) und ist separat kostenlos über die Sun Java-Homepage beziehbar.

1.4 Vom Quellcode zum lauffähigen Programm

Die Programmentwicklung vollzieht sich im klassischen edit-build-run-Zyklus. Als Besonderheit generiert der Java-Compiler je eine Bytecode-Datei (Dateiextension `.class`) für jede zugreifbare Klasse innerhalb der Quelldatei.

Die entstehenden Class-Dateien werden durch die Laufzeitumgebung interpretativ ausgeführt.

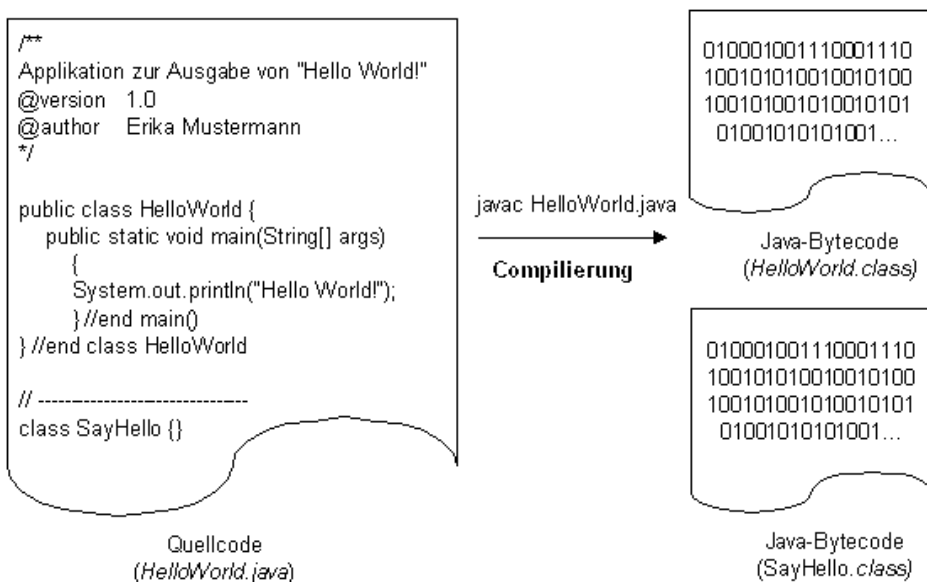


Abbildung 5: Der Compilierungsvorgang (click on image to enlarge!)

Im Beispiel enthält die Quellcodedatei [HelloWorld.java](#) die beiden Klassendateien [HelloWorld](#) und [SayHello](#).

Diejenige Klasse, welche die Main-Methode beinhaltet muß identisch der sie enthaltenden Quelldatei benannt sein -- im Beispiel HelloWorld.

Mit `javac HelloWorld.java` wird die Datei übersetzt, und die beiden Class-Dateien HelloWorld.class und SayHello.class erzeugt.

Die Programmausführung erfolgt durch Absetzen von `java HelloWorld` auf der Kommandozeile. (Achtung! Keine Dateierstension angeben!)

Der Aufruf `java SayHello` führt hingegen wegen des Fehlens der Main-Methode in der Klasse SayHello zur Fehlermeldung `Exception in thread "main" java.lang.NoSuchMethodError: main`.

▲ 2. Syntax und Semantik der Programmiersprache Java

▲ 2.1 C, C++, C# und Java

Wie bereits in der [Einführung angedeutet](#) wurde Java nahe an der Syntax der verbreiteten hybrid objektorientiert-prozeduralen Sprache C++ entwickelt. Jedoch mit der Einschränkung, deren mitunter kryptischen Syntax deutlich zu vereinfachen. Überdies wurde auf wesentliche dort anzutreffende Sprachmerkmale verzichtet.

Die [Java Language Specification](#) führt bereits im Vorwort aus, daß sich die Java-Sprachentwickler zwar in einer Vielzahl von Punkten an C und C++ orientierten, jedoch der Sprachaufbau stark von diesen Beiden Vätern abweicht. Aus praktischer Motivation heraus wurde beim Sprachdesign auf die Einführung neuer und ungetesteter Sprachelemente verzichtet.

Wie [erwähnt](#) ist Java als streng typisierte Sprache ausgelegt. Daher können die meisten Typfehler bereits zur Übersetzungszeit erkannt werden. Technisch gesehen meint strenge Typisierung (auch *statische* Typisierung), daß der Typ einer Variable während der Programmausführung nicht verändert werden kann. Durch statische Typanalyse während des Compilierungsvorganges können Typfehler erkannt werden. Durch polymorphe Aufrufe kann es jedoch auch während der Programmausführung zu Typfehlern kommen, da hierbei der dynamische Typbindung zum Einsatz kommt.

Anders als C und C++ verfügt Java über keinerlei systemnahe Konstrukte. Direkte Hardwarezugriffe und Systemprogrammierung im Allgemeinen kann daher nicht realisiert werden.

Die Sprachväter begründen dies mit der Transparenz einer *high-level*-Sprache, die keinerlei Rückschlüsse auf die darunterliegende Hardware zulassen sollte.

Anders als C/C++ verfügt Java über [automatische Speicherbereinigung](#) (engl. *garbage collection*), welche die fehlerträchtigen Speicheroperationen (insbesondnere *free* und *delete*) obsolet werden läßt. Konsequenterweise läßt Java die Allokation beliebiger Speicherbereiche nicht zu. Nicht mehr

benötigte Speicherplätze (Variablen) können zwar durch den Programmierer (durch die explizite Zuweisung von NULL) als nicht mehr benötigt gekennzeichnet werden, Freigabefunktionen stellt die Sprache jedoch nicht zur Verfügung.

Bekannte, und berüchtigte, Fehlerquellen wie Arrayzugriffe ohne vorherige Indexprüfung, variabel lange Parameterlisten oder Zeiger nebst Zeigerarithmetik existieren nicht, und verleihen dem Sprachentwurf dadurch zusätzliche Sicherheit. Der Verzicht auf (explizite) Zeiger zieht die Behandlung aller Methodenparameter als Werte (call-by-value) nach sich. Ebenso wurde auf Umgebungseigenschaften wie den Präprozessor und Includedateien vollständig verzichtet wurde. Die Strukturierung des Java-Quellcodes kann über ein eigenes Paketkonzept erfolgen.

Nützlichkeiten wie die Lockerung des Variablendefinitionszwanges am Blockanfang wurden beibehalten. Selbiges gilt für die Aufhebung der Trennung zwischen Deklaration und Definition bei einfachen Variablen, wie sie bereits in anderen Sprachen verwirklicht ist. Für Objekte existiert diese Trennung -- sinnvollerweise -- weiterhin fort.

Hinsichtlich objektorientierter Konzepte geht Java deutlich über C++ hinaus. Dergestalt, daß weder klassenlos Programme entwickelt werden können, noch Structs und Unions als Zwitter zwischen Variablen und Klassen implementiert sind. Die Prämisse strengerer Objektorientierung erklärt auch das Verbot globaler Variablen und Methoden. Allerdings sind die primitiven skalaren Datentypen (wie `int`, `char`, etc.) nicht als Objekte realisiert.

Analog C++ können Methodennamen überladen werden (Ad-hoc Polymorphie). Zur Eindeutigkeitsidentifikation wird die Signatur (gebildet aus Methodennamen und übergabeparametern) herangezogen. Die Überladung von Operatoren, wie in C++ möglich, ist nicht vorgesehen.

Der in C++ realisierte Vererbungsmechanismus wurde unter der Einschränkung übernommen, Mehrfachvererbung zu verbieten. Um trotz dieser Restriktion sinnvolle Anwendungsentwicklung betreiben zu können wurde Java als zusätzliches Sprach-Konzept die Schnittstelle (engl. *Interface*) hinzugefügt. Hiervon können eine Klasse beliebig viele implementiert werden.

Im Gegensatz zu C++ erben Klassen ohne ausdrücklich angegebene Elternklasse automatisch per Vorgabe von [java.lang.Object](#).

Parametrische Polymorphie in Form von Templates, wie in C++ anzutreffen, ist in Java erst ab der Sprachversion 1.5 realisiert.

Von C++ wurde die Fehlerbehandlung in Form von Ausnahmen (engl. *exceptions*) übernommen. Hierdurch können Fehlerereignisse zentralisiert behandelt werden. Zusätzlich wird der Kontrollfluß von Fehlerbehandlungscode bereinigt und dadurch übersichtlicher.

Offensichtlich ist die Übernahme des Kommentierungsstils von C und C++. So können alle Kommentarkonstrukte wie in den vorgenannten Sprachen üblich eingesetzt werden. Zusätzlich wird ein separater Kommentarstil (`/**...*/`) für Quellen automatisierter Dokumentation angeboten. Zwischen den so abgegrenzten Regionen können vorgegebene Marken platziert werden; diese werden beispielsweise durch das JDK-Werkzeug [javadoc](#) verarbeitet.

Augenfälligstes abgrenzendes Merkmal von Java gegenüber C/C++ ist es, daß kein plattformspezifischer Maschinencode als Resultat des Compilierungsvorganges erzeugt wird, sondern binärer Bytecode genannte Zwischenrepräsentation. Diese wird durch die Java Virtual Machine zur Ausführungszeit interpretativ abgearbeitet.

Anmerkung: Bytecode ist nicht Java-spezifisch, sondern kann auch durch andere Sprachen erzeugt werden, was mit unter auch geschieht.

Microsofts C#

Die durch Microsoft entwickelte Sprache C# (sprich: *C sharp*) weist einige interessante Parallelen zu Java auf, führt jedoch auch neue Konzepte ein.

Jenseits der Einordnung in die Komponentenarchitektur der .NET-Plattform, welche nur schwer mit der der Java-Plattform (insbesondere der J2EE-Plattform) verglichen werden kann, stellt C# jedoch eine vollwertige -- sehr stark an Java orientierte -- Programmiersprache dar. Bereits das, zu erwartende, Standardbeispiel der HelloWorld-Applikation läßt die enge Verwandtschaft, abzulesen an der Syntax deutlich werden (Vergleiche HelloWorld [in Java](#), in [in C#](#)).

Im Gegensatz zu C# ist Java auf der Programmierenebene nicht *rein* objektorientiert wie beispielsweise SmallTalk. So treten auch hier die primitiven Datentypen als nicht-objektartige Werte auf.

Ebenso wie in Java ist ausschließlich Einfachvererbung zugelassen, ergänzt wird diese um Schnittstellen. Auch C# erlaubt es einer Klasse mehrere Schnittstellen zu implementieren. Die Syntax unterscheidet jedoch nicht mehr explizit zwischen Schnittstellenimplementierung und Erben von einer Klasse (siehe [Beispiel](#)). Unverändert zu Java wird auch jede Klasse, die über keine Elternklasse verfügt automatisch als Subklasse von `object` eingeordnet. Blattknoten einer Vererbungshierarchie (d.h. Klassen für die durch den Programmierer vorgegeben wird, sie sollen nicht weiter durch Ableitung spezialisiert werden) können mit dem Schlüsselwort `sealed` -- in Java: `final` -- gekennzeichnet werden.

Zusätzlich zu Klassen beinhaltet C#, wieder, den Sprachmechanismus der Struktur (`struct`). Hierüber wird die Differenzierung zwischen Wert- und Referenztypen realisiert. Während Klassen, Java-konform Referenztypen bezeichnen, übernehmen Strukturen die der Werttypen. Desweiteren werden Structs durch den Compiler als Bestandteile umgebender Objekte übersetzt. Durch diese Charakteristika unterscheiden sich C#-Structs stark von den dortigen Klassenstrukturen, weshalb sie auch nicht analog zu deren Schema verwirklicht sind. Strukturen können weder erben noch vererbt werden, lediglich die Möglichkeit Schnittstellen zu implementieren bleibt erhalten.

Deutlich komfortabler als in Java fällt jedoch der Umgang mit Referenz- und Werttypen aus. Das in C# angebotene *boxing* und *unboxing* übertrifft deutlich die Benutzungsfreundlichkeit der Java-Wrapperklassen ([Beispiel](#)). Dieser Mißstand wurde jedoch in der Javasprachversion 1.5 behoben, die dieses Konzept ebenfalls einführt.

Erstmals reichert C# auch den Sprachumfang C/C++-basierter Sprachen um neue syntaktische Konstrukte an. Hierunter fallen beispielsweise Schlüsselworte zur Definition von Delegationsobjekten, die als solche explizit im Programmcode deklariert werden können.

Wie Java compiliert auch C# in eine Zwischenrepräsentation. Allerdings mit dem Unterschied, daß diese nicht interpretativ abgearbeitet wird, sondern erneut in maschinenspezifisches natives Format übersetzt wird.

Die Tabelle stellt einige Sprachmerkmale von C++, Java und C# gegenüber

(Tabelle in Anlehnung an: Eisenecker, U.: Dissonanz oder Wohlklang, in: iX 9/2000, Hannover, 2000, p. 48-51)

Sprachmerkmal	C++	Java	C#
Automatische Speicherbereinigung (garbage collection)			
Coercion (Typumwandlungspolymorphie)			
Globale Methoden			
Inklusionspolymorphie			
Operatoroverloading			
Parametrische Polymorphie (Templates)		 (1)	 (2)
Referenztypen			
Überladungspolymorphie			
Werttypen			 <i>unified type system</i>
Zeiger			

(1) Ab Version 1.5 für Klassen der Collection API im Sprachumfang enthalten.

(2) Die Aufnahme in die Sprache ist für die Nachfolgeversion des .NET-Frameworks 1.1 geplant. Einen Ausblick auf die geplanten Sprachmerkmale liefert das Projekt [CLRGEN](#) von Microsoft Research.

▲ 2.2 Grundstrukturen

▲ 2.2.1 Programmaufbau

Bereits am [HelloWorld-Beispiel](#) wird der wesentliche Aufbau einer Javaquellcodedatei sichtbar.

```
(1)public class Minimal {
(2)    public static void main(String[] args) {
(3)        //..
(4)    } //main()
(5)} //class Minimal
```

Beispiel 2: Minimales Java-Programm

Zunächst erfolgt die Definition einer Klasse; im Beispiel `Minimal`.

Vor dem Schlüsselwort `class` ist die Sichtbarkeit auf `public` festgelegt, was eine allgemeine Sichtbar- und Zugriffsbarkeit der Klasse bewirkt.

Im Gegensatz zu C++ ist jedoch die Klassendefinition zwingend erforderlich! Es ist nicht möglich Javaquellcode, der keine Klassendefinition enthält, zu übersetzen. Hingegen ist es ohne weiteres möglich gewöhnliche C-Programme mit einem C++-Compiler zu übersetzen.

Darüberhinaus ist es zwingend vorgeschrieben die Quellcodedatei identisch zur `public` deklarierten Klasse zu benennen.

Eine weitere Besonderheit gegenüber C++ stellt die Platzierung der `main`-Funktion dar. Während diese in C++, selbst bei der Verwendung von Klassen ([siehe Beispiel \(HelloWorld.cpp\)](#)), außerhalb jeder Klasse angeschrieben werden muß, um automatisch beim Programmstart ausgeführt zu werden, erzwingt Java die `main`-Methode innerhalb einer `public` deklarierten Klasse.

Die Signatur der `main`-Methode ist mit `public static void main(String[] args)` fixiert. Davon Abweichende Spezifikationen sowohl in Rückgabewert als auch Parameterliste, können zwar in Bytecode übersetzt werden, jedoch wird diese abweichende `Main`-Methode nicht mehr automatisch durch das Laufzeitsystem aufgerufen.

Auch hier zeigt sich Java deutlich restriktiver als C/C++, die beide beliebige Rückgabetypen und -- in Grenzen -- leicht variierende Parameterlisten zulassen.

Abschließend: Innerhalb der `main`-Methode der `public` deklarierten Klasse kann der auszuführende Code angegeben werden.

▲ 2.2.2 Einfache Datentypen

Wie herkömmliche prozedurale Programmiersprachen auch stellt Java primitive Datentypen zur Verfügung. Im Gegensatz zu manchen etablierten objektorientierten Programmiersprachen (z.B. SmallTalk) sind diese Datentypen in Java intern nicht als Objekte realisiert.

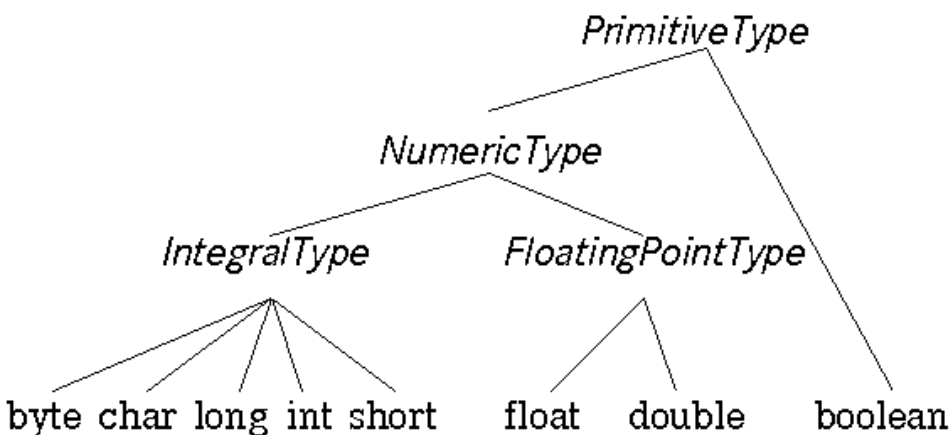


Abbildung 6: Das Java-Typsystem (click on image to enlarge!)

Im Einzelnen werden angeboten:

Datentyp	Erklärung	Wertebereich
boolean	Wahrheitswert	true oder false
char	Einfaches Zeichen aus dem 16-Bit Unicode Zeichensatz	
byte	8-bittige vorzeichenbehaftete Ganzzahl	-2^7 bis 2^7-1 $-128 \leq \text{byte} \leq 127$
short	16-bittige vorzeichenbehaftete Ganzzahl	-2^{15} bis $2^{15}-1$ $-32768 \leq \text{short} \leq 32767$
int	32-bittige vorzeichenbehaftete Ganzzahl	-2^{31} bis $2^{31}-1$ $-2147483648 \leq \text{int} \leq 2147483647$
long	64-bittige vorzeichenbehaftete Ganzzahl	-2^{63} bis $2^{63}-1$ $-9223372036854775808 \leq \text{long} \leq 9223372036854775807$
float	32-bittige Gleitkommazahl (nach IEEE 754-1985)	Größtmögliche positive Zahl: $3.40282347\text{e}+38\text{f}$ Kleinstmögliche positive Zahl: $1.40239846\text{e}-45\text{f}$
double	64-bittige Gleitkommazahl (nach IEEE 754-1985)	Größtmögliche positive Zahl: $1.79769313486231570\text{e}+308$ Kleinstmögliche positive Zahl: $4.94065645841246544\text{e}-324$
Object	Objektwertiger Datentyp. Jedes Objekt hat (implizit) diesen Typ.	

Siehe [Java Language Specification](#)

Anmerkungen:

- Der Wahrheitswert ist nicht wie in C++ üblich als Ganzzahl realisiert. Auch eine explizite Typumwandlung ist nicht möglich, und wird zur Übersetzungszeit (als Typfehler) abgefangen (Compilerfehlermeldung: `inconvertible types`). Hinweis: Zwar ist im ANSI-C++-Standard ein `bool` als expliziter Datentyp (üblicherweise der Länge 1 Byte) definiert, der jedoch die Wahrheitswerte als Integer-Zahlen ablegt. So gilt: `true == 1` bzw. `false == 0` ([siehe Beispiel](#))
- Die Ganzzahldarstellung erfolgt immer im Zweierkomplement, weshalb der positive Wertebereich eine Zahl weniger als der negative umfaßt.
- Alle numerischen Typen sind vorzeichenbehaftet. Vorzeichenlose Pendant (wie in C durch Voranstellen eines *unsigned* bildbar) existieren nicht.
- Die Bitlänge aller Datentypen ist festgelegt, und nicht plattformabhängig variierbar.
- Für die beiden Gleitkommatypen existiert ein Literal zur Darstellung von Plus- und Minus-Unendlich. Zusätzlich ein mit NaN (*Not a Number*) ein ausgezeichnete Wert für nicht interpretierbare Belegungen.
- `char` ist zwar als Zeichensymbol ausgelegt, kann aber auch in mathematischen Termen anstelle von `short` verwendet werden; hier wird `char` jedoch vorzeichenlos(!) interpretiert. Die Konversion von `char` zu `int` erfolgt implizit, die Umgekehrung bedarf jedoch einer ausdrücklichen Typumwandlung.
- Die nicht standardkonforme Implementierung der virtuellen Maschine von Microsoft bietet mit `object` (32-bittige Referenz auf ein Java-Objekt) und `returnAddress` (32-Bit) zwei zusätzliche Datentypen an.
- Ebenso wie in C/C++ üblich können Konstanten durch spezielle Muster typisiert werden.
 - `\uxxxx` für Unicodezeichen (`xxxx` ist die hexadezimale Zeichennummer)
 - `...l` oder `...L` für Longwerte
 - `0...` für Oktalwerte
 - `0x...` für Hexadezimalwerte
 - `...f` oder `...F` für Floatwerte
 - `...d` oder `...D` für Doublewerte
- Für die primitiven built-in Typen wird keine über- oder Unterlauferkennung angeboten. ([Siehe Beispiel Overflow.java](#)).
- Auffallend ist, daß keine Arraytypen existieren. Diese sind in Java nicht als Aggregation primitiver built-in Typen realisiert, sondern als Klasse der API umgesetzt.


```

(1)public class ConstFormats {
(2)    public static void main(String[] args) {
(3)        int i = 052;           //octal
(4)        System.out.println("i as decimal = "+i);
(5)
(6)        i = 0x2a;              //hexadecimal
(7)        System.out.println("i as decimal = "+i);
(8)
(9)        long l = 0x2aL; //hexadecimal long
(10)       System.out.println("l as decimal = "+l);
(11)
(12)       float f = 0x2af;        //hexadecimal float
(13)       System.out.println("f as decimal = "+f);
(14)    } //main()
(15)} //class ConstFormats

```

Beispiel 3: Verschiedene Deklarationen und Wertebelegungen [ConstFormats.java](#)

Das Programm liefert folgende Ausgabe:

```

i as decimal = 42
i as decimal = 42
l as decimal = 42
f as decimal = 687.0

```

VORSICHT! Die hexadezimale Definition des Floatwertes liefert nicht -- wie vielleicht intuitiv zu erwarten -- 42 als ganzzahligen Anteil, sondern die Gleitkommainterpretation des hexadezimal spezifizierten Bitmusters.

Als streng typisierte Sprache muß jede Variable in Java mit einem Typ deklariert werden, ungetypte Variablen existieren nicht.

Nach der Deklaration in einem Programm wird der Variableninhalt auf einen reservierten, für den Programmierer nicht zugänglichen Wert *undefined* gesetzt. Hierdurch kann der Compiler lesende Referenzierungen vor Wertdefinition zur Übersetzungszeit erkennen.

Die Deklaration geschieht, angelehnt an C/C++ durch Angabe des Typs gefolgt durch den Variablennamen. Optional kann dieses Statement durch die Festlegung eines Vorgabewertes ergänzt werden.

```

(1)public class NotInitialized {
(2)    public static void main (String[] args) {
(3)        int i;
(4)        System.out.println("i= "+i); //i is not initialized yet
(5)    } //main()
(6)} //class notInitialized

```

Beispiel 4: Nicht initialisierte Referenzierung [NotInitialized.java](#)

Liefert beim Übersetzen die Fehlermeldung:

```

NotInitialized.java:6: variable i might not have been initialized
    System.out.println("i= "+i); //i is not initialized yet
                        ^
1 error

```

```

(1)public class CharArithmetic {
(2)    public static void main (String[] args) {
(3)        char c = 'a';
(4)
(5)        System.out.println("c = "+c);
(6)        c++;
(7)        System.out.println("c = "+c);
(8)
(9)        int i = c;
(10)
(11)       System.out.println("i = "+i);
(12)       System.out.println("i = "+ (char) i);
(13)    } //main()

```

```
(14)} //class CharArithmetic
```

Beispiel 5: Verwendung von char als numerischer Typ [CharArithmetic.java](#)

Das Programm liefert bei der Ausführung folgende Ausgabe:

```
c = a
c = b
i = 98
i = b
```

▲ 2.2.3 Operatoren

Java bietet 37 verschiedene Operatoren an, die ihrer Semantik nach mit denen von C/C++ weitestgehend identisch sind ([siehe Language Specification](#)).

Je nach Typ auf dem Operator definiert ist, wird unterschieden zwischen: Integralen-, Fließkomma-, Boole'schen- und objektwertigen-Operatoren.

▲ Operatoren auf integralen Typen (

**siehe Java API
Specification)**

- Vergleichsoperatoren, die als Resultat boolean liefern:
 - numerischer Vergleich: <, <=, > und >= ([siehe Java Language Specification](#))
 - numerische Gleichheit: == und != ([siehe Java Language Specification](#))
- Numerische Operatoren, die ganzzahligen Wert (int oder long) zurückliefern:
 - unäres Plus und Minus ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).
 - multiplikative Operatoren *, / und % ([siehe Java Language Specification](#)).
 - additive Operatoren + und - ([siehe Java Language Specification](#)).
 - inkrement Operatoren (Prefix- und Postfixvariante) ++ ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).
 - dekrement Operatoren (Prefix- und Postfixvariante) -- ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).
 - Shift Operatoren <<, >> bzw. >>> (Einführung von Füllnullen) ([siehe Java Language Specification](#)).
 - So gilt: -2 >>> 2 = -1 aber: -2 >> = 1073741823
 - bitweises Komplement ~ ([siehe Java Language Specification](#)).
 - bitweise Integeroperatoren &, | und ^ ([siehe Java Language Specification](#)).
- Konditionaloperator (Kurzschreibweise für if) ? : ([siehe Java Language Specification](#)).
- explizite Typumwandlung (cast), zur Umwandlung jedes Integraltypen in einen beliebigen numerischen ([siehe Java Language Specification](#), [siehe Java Language Specification](#)).

Diese built-in Operatoren nehmen keinerlei Fehlerprüfung oder -meldung vor. Lediglich die beiden Integerdivisionsoperatoren / und % werfen im Fehlerfalle eine ArithmeticException-Ausnahme falls der Divisor gleich Null ist ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).

▲ Operatoren auf Fließkommatypen (

**siehe Java
Language
Specification)**

Auf diesen Typen sind die auch auf integralen Typen zugelassenen arithmetischen-, numerischen Vergleichs-, Inkrement- und Dekrement-Operatoren verfügbar. Ferner existiert der numerische Cast ([siehe Java API Specification](#)).

Auch die Fließkommaoperatoren lösen keine Exceptions aus. überlauf wird als positiv Unendlich, Unterlauf entsprechend als negativ Unendlich dargestellt. Liefert die Operation kein mathematisch interpretierbares Resultat, wird der Wert auf NaN gesetzt. In der Konsequenz resultiert auch NaN falls ein so gesetzter Operand erneut verknüpft wird. Dies gilt nicht für die Gleichheitsoperation.

- Vergleichsoperatoren, die als Resultat boolean liefern:
 - numerischer Vergleich: <, <=, > und >= ([siehe Java Language Specification](#))
 - numerische Gleichheit: == und != ([siehe Java Language Specification](#))
- Numerische Operatoren, die ganzzahligen Wert (float oder double) zurückliefern:
 - unäres Plus und Minus ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).
 - multiplikative Operatoren *, / und % ([siehe Java Language Specification](#)).
 - additive Operatoren + und - ([siehe Java Language Specification](#)).
 - inkrement Operatoren (Prefix- und Postfixvariante) ++ ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).
 - dekrement Operatoren (Prefix- und Postfixvariante) -- ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).
- Konditionaloperator (Kurzschreibweise für if) ? : ([siehe Java Language Specification](#)).
- explizite Typumwandlung (cast), zur Umwandlung jedes Integraltypen in einen beliebigen numerischen ([siehe Java Language Specification](#), [siehe Java Language Specification](#)).

Anmerkung: Identisch zu ANSI C/C++ wurde der unäre Operator + lediglich aus Symmetriegründen zum unären eingeführt.

▲ Operationen auf Boole'schen Typen

Auf Boole'schen Typen sind alle relationalen und logischen Operatoren zugelassen.

Als Operand des Konditionaloperators können neben Boole'schen Werten auch Integralwerte angegeben werden. Diese werden gemäß C-Konvention zu true konvertiert sofern sie ungleich Null sind, andernfalls zu false.

- relationale Operatoren (Gleichheit und Ungleichheit) == und != ([siehe Java Language Specification](#)).
- logisches Komplement ! ([siehe Java Language Specification](#)).
- logische Operatoren &, ^ und | ([siehe Java Language Specification](#)).
- konditionelles-und und konditionelles-oder && bzw. || ([siehe Java Language Specification](#) bzw. [siehe Java Language Specification](#)).

Die Auswertung erfolgt im sog. short circuit-Verfahren. Hierbei wird nicht zwingend der gesamte Ausdruck ausgewertet. Bei der *Ver-und-ung* wird abgebrochen, sobald ein falsches Ergebnis vorliegt; bzw. bei der *Ver-oder-ung* sobald ein wahres Ergebnis vorliegt.

- Konditionaloperator ? : ([siehe Java Language Specification](#))

▲ Operatoren auf objektwertigen Typen

- + zur Stringkonkatenation. Ist einer der beiden angegebenen Operaden nicht vom Typ String, so wird zur Laufzeit eine Stringkonversion durchgeführt ([siehe Java Language Specification](#)). Diese Stringkonversion ist immer möglich, da alle Objekte von [java.lang.Object](#) die Methode toString erben.

Anmerkung: hierbei handelt es sich strenggenommen um einen überladenen Operator

- Abfragen auf den Typ eines Objekts können durch den instanceof-Operator erledigt werden. Er liefert einen Wahrheitswert zurück.
Syntax: *object instanceof class*.

▲ Schlussbemerkungen

- Wie in C/C++ auch existieren die abkürzenden Zuweisungsschreibweisen: +=, -=, *=, /=, &=, |=, ^=, %=, <<=, >>=, >>>=.
- Die aus C/C++ bekannten Speicherzellen-bezogenen Operatoren * und & werden wegen des

Fehlens expliziter Speicherreferenzen nicht unterstützt.

- Ebenso der sizeof-Operator. Für ihn existiert kein Anwendungsfall, da einerseits die Größe der built-in Typen festgelegt ist, andererseits keine Speicherblöcke wahlfrei allokiert werden können.
- Die Operanden aller diskutierten Operatoren, außer `&&`, `||` und `? :`, werden vor der Ausführung des Operators ausgewertet.

Dies gilt insbesondere für Operatoren die eine Exception auslösen können. Eine solche wird ggf. nach Auswertung aller Operaden erzeugt.

▲ Operatorpräzedenz

Es gelten folgende Operatorpräzedenzen (geordnet von oben (entspricht höchster Präzedenz) nach unten (niedrigster Präzedenz)):

Operator	Symbol
Postfix-Operatoren	<code>[] . (params) expr++ expr--</code>
unäre Operatoren	<code>++expr --expr +expr -expr ~ !</code>
Erzeugung oder Typumwandlung	<code>new (type)expr</code>
Multiplikationsoperatoren	<code>* / %</code>
Additionsoperatoren	<code>+ -</code>
Verschiebeoperatoren	<code><< >> >>></code>
Vergleichsoperatoren	<code>< > <= >= instanceof</code>
Gleichheitsoperatoren	<code>== !=</code>
Bitoperator Und	<code>&</code>
Bitoperator exklusives Oder	<code>^</code>
Bitoperator inklusives Oder	<code> </code>
logisches Und	<code>&&</code>
logisches Oder	<code> </code>
Konditionaloperator	<code>? :</code>
Zuweisungsoperatoren	<code>= += -= *= /= %= >>= <<= >>>= &= ^= =</code>

▲ Automatische Typkonversion

Die automatische Typkonversion wird durch den Compiler bzw. das Laufzeitsystem immer dann angewandt, wenn nicht alle Operanden typgleich sind. Dies ist beispielsweise bei der Multiplikation einer Int-Zahl mit einem Fließkommawert der Fall.

Die automatische Typumwandlung ist im Wesentlichen identisch zur in C/C++ realisierten ausgelegt.

	byte	char	short	int	long	float	double	boolean
byte	int	int	int	int	long	float	double	
char		int	int	int	long	float	double	
short			int	int	long	float	double	
int				int	long	float	double	

long	long	float	double	
float		float	double	
double			double	
boolean				boolean

Die Tabelle stellt die automatischen Typkonversionen zur Festlegung des Ausdruckstyps dar. Da alle Operatoren kommutativ sind, ist die Tabelle symmetrisch zur Hauptdiagonale (nur die obere Dreiecksmatrix ist aus übersichtlichkeitsgründen ausgefüllt).
Alle Typkonversionen werden statisch und operatorunabhängig durchgeführt, d.h. mögliche Typfehler werden zur Compilezeit erkannt und gemeldet.

▲ 2.3 Kontrollstrukturen

▲ 2.3.1 Selektion und Mehrfachselektion -- das if und case-Statement

Das if-Statement ist in Java analog der aus C/C++ bekannten Semantik und Syntax realisiert ([Siehe Language Specification](#)):

```

IfThenStatement:
    if ( Expression ) Statement

IfThenElseStatement:
    if ( Expression ) StatementNoShortIf else Statement

IfThenElseStatementNoShortIf:
    if ( Expression ) StatementNoShortIf else StatementNoShortIf

```

```

(1)public class IfTest {
(2)    public static void main(String[] args) {
(3)        int i = 42;
(4)
(5)        if (1==1)
(6)            if (i < 50)
(7)                i++;
(8)            else
(9)                i--;
(10)    } //main()
(11)} //end class IfTest

```

Beispiel 6: If-Statement mit dangling else [IfTest.java](#)

Leider wurde in Java die Chance zur Ausmerzung des lästigen und fehlerträchtigen *dangling else-Problems* nicht genutzt. Daher wird -- konform zu C/C++ -- ein *else* immer dem letzten vorhergehenden *if* zugeordnet.

Strenger im Vergleich zu C/C++ ist hingegen die Typisierung der Expression als Bedingung innerhalb der Verzweigung gefaßt. Hier ist zwingend der Typ `boolean` erforderlich.

Auch das `switch`-Statement ist identisch zum C/C++-Analogon realisiert. Ebenso wie dort müssen die Alternativzweige mit einem expliziten `break` abgeschlossen werden. Innerhalb der `case`-Verzweigungen sind nur konstante Ausdrücke der Typen `char`, `byte`, `short` oder `int` zugelassen. ([siehe Java Language Specification](#)).

Syntax:

```

SwitchStatement:
    switch ( Expression ) SwitchBlock

```

```

SwitchBlock:
    { SwitchBlockStatementGroupsoptional SwitchLabelsoptional }

SwitchBlockStatementGroups:
    SwitchBlockStatementGroup
    SwitchBlockStatementGroups SwitchBlockStatementGroup

SwitchBlockStatementGroup:
    SwitchLabels BlockStatements

SwitchLabels:
    SwitchLabel
    SwitchLabels SwitchLabel

SwitchLabel:
    case ConstantExpression :
    default :

```

Beispiel eines switch-Statements:

```

(1) public class SwitchTest {
(2)     public static void main(String[] args) {
(3)         int i = 42;
(4)
(5)         here:
(6)         switch (i) {
(7)             case 0: i++;
(8)                                     break here;
(9)             case 1:
(10)            case 2: i--;
(11)                                     break;
(12)            default:
(13)                                     i *= 10;
(14)        } //switch
(15)    } //main()
(16)} //class SwitchTest

```

Beispiel 7: Switch-Statement [SwitchTest.java](#)

Anmerkungen:

- Der break-Anweisung kann eine Marke nachgestellt werden, die angibt auf welcher Schachtelungsebene fortgesetzt werden soll ([siehe Java Language Specification](#)). Hierbei sind jedoch nur Rückwärtsreferenzen zugelassen.
- Soll ein case-Körper für mehrere Alternativen gelten, so müssen diese explizit mit einleitendem case aufgeführt werden. Pro case ist jeweils nur genau eine konstante Bedingung zugelassen.
- Obwohl switch-Statement syntaktisch unverändert von C/C++ übernommen wurde, existieren in Java zusätzliche Restriktionen, welche die Kompatibilität einschränken. So muß der Rumpf einer switch-Anweisung die zugehörigen case-Alternativen direkt enthalten; Konstruktionen wie *duff's devices* ([siehe Java Language Specification](#)) sind daher nicht compilierbar.
- Die switch-Bedingung muß vom Typ byte, short, char oder int sein.

▲ 2.3.2 Iteration -- for-, do-while-Schleifen

Java bietet die bereits in C/C++ eingeführten Schleifenkonstrukte

- for ([siehe Java Language Specification](#))
- while ([siehe Java Language Specification](#))
- do ([siehe Java Language Specification](#))
- break und continue ([siehe Java Language Specification](#))

an.

Die **for-Struktur** besteht aus drei optionalen Komponenten: Initialisierung(en), Fortsetzungsbedingung(en) und Wertaktualisierung(en) (auch Reinitialisierung(en)). Ebenso wie in C/C++ sind diese durch Semikola voneinander abgetrennt.

Im Initialisierungs- und Fortsetzungsteil sind mehrere, durch Komma separierte, Ausdrücke

zuglassen. Im Fortsetzungsbedingungsteil hingegen nicht! Hier müssen mehrere Bedingungen durch logisches *Und* (&&) verbunden werden.

```
(1)public class ForTest {
(2)    public static void main(String[] args) {
(3)        for (int i=1, j=-2; i <= 10 && j <=0 ; i+=2, j++)
(4)            System.out.println("i = "+i+"  j = "+j);
(5)    } //main()
(6)} //class ForTest
```

Beispiel 8: Eine for-Schleife [ForTest.java](#)

Alle Schleifen können vorzeitig, d.h. trotz gültiger Fortsetzungsbedingung(en) wahlfrei mit der break-Anweisung verlassen werden.

Generell versucht die break-Anweisung hinter dem nächstliegenden (d.h. innsteren erreichbaren) schließenden Schleifenkonstrukt fortzusetzen. Der Einsatz dieser Anweisung außerhalb der beschriebenen Strukturen wird per Übersetzungsfehler verhindert.

Zusätzlich können Sprungziele durch Marken explizit benannt werden. Diese *labels* werden durch einen eindeutigen Namen abgeschlossen von einem Doppelpunkt symbolisiert.

Es sind jedoch nur „Rückwärtssprünge“ möglich. ([siehe Java Language Specification](#))

```
(1)public class BreakTest {
(2)    public static void main(String[] args) {
(3)        myLabel:
(4)        for (int i=1; i<=100; i++) {
(5)            for(int j=1; j<=100; j++) {
(6)                if (i*j >= 42) {
(7)                    System.out.println("exiting loop");
(8)                    break myLabel;
(9)                } //if
(10)            } //for
(11)        } //for
(12)        System.out.println("continuing...");
(13)    } //main()
(14)} //class BreakTest
```

Beispiel 9: Verlassen einer Schleife mit break [BreakTest.java](#)

Im Gegensatz zu vielen prozeduralen Programmiersprachen verfügt Java über kein goto-Statement welche beliebige Sprünge erlauben würde.

Allerdings scheint beim Sprachdesign durchaus an diese Möglichkeit gedacht worden zu sein. Findet sich doch goto in der Aufzählung der reservierten Schlüsselworte ([siehe Java Language Specification](#)) und im Index ([siehe Java Language Specification](#)).

Die Referenzimplementierung des Javacompilers von SUN nutzt das Schlüsselwort lediglich um den illegalen Beginn eines Ausdrucks zum Übersetzungszeitpunkt anzuzeigen.

Zum vorzeitigen Rücksprung aus dem Schleifenkörper ist das das continue-Statement definiert. Nach seiner Ausführung erfolgt unverzüglich die Auswertung der Fortsetzungsbedingung, unter Auslassung aller folgenden Anweisungen des aktuellen Blocks.

Analog der break-Anweisung kann die Schachtelungstiefe in der fortgefahren werden soll durch Angabe eines Labels gesteuert werden. ([siehe Java Language Specification](#))

Kopf- und Fußgesteuerte Schleifen, werden mit der while ... do-Konstruktion analog zu C/C++ realisiert.

Auch hier wird der Typ der Bedingung bereits zur Übersetzungszeit auf boolean geprüft.

Häufige Anwendungsform von Schleifen ist die Traversierung einer Objektmenge.

Das nachfolgende Beispiel zeigt die „klassische“ Vorgehensweise zur Ausgabe aller Elemente einer Aufzählung. Hierzu wird zunächst Elementanzahl ermittelt und anschließend in einer for-Schleife, beginnend mit der kleinsten Indexnummer (0) bis zur ermittelten Obergrenze inkrementiert. An jeder Indexposition wird das unter dieser Ordnungsnummer abgelegte Element ausgegeben. Der Vorgehensweise unterliegt den beiden Grundannahmen, daß die Werte einerseits kontinuierlich (d.h. keine Indexposition ist unbesetzt) abgelegt sind. Und zweitens, daß die Wertemenge ab der Indexposition 0 aufsteigend abgelegt ist. Beide Annahmen können für die durch die Java-Standardklasse [Vector](#) realisierte Aufzählung als erfüllt angenommen werden.

```

(1)import java.util.Vector;
(2)
(3)public class NaiveIteration {
(4)    public static void main(String[] args) {
(5)        Vector v = new Vector();
(6)        v.add(new String("Berta"));
(7)        v.add(new String("Anna"));
(8)        v.add(new String("Cäsar"));
(9)
(10)        for (int i=0; i<v.size(); i++) {
(11)            System.out.println(v.get(i));
(12)        } //for
(13)    } //main()
(14)} //class NaiveIteration

```

Beispiel 10: Naive Traversierung einer Objektmenge [NaiveIteration.java](#)

Die Lösung leistet zwar das gewünschte, jedoch wirkt die Schleifenkonstruktion unnötig komplex. Überdies zwingt sie den Programmierer Daten abzuspeichern (in Form des Schleifenzählers `i` sowie der implizit angeforderten unbenannten Speicherstelle zur Ablage der Elementanzahl), die nur zum Zweck der Mengentraversierung benötigt werden.

Um die Umsetzung dieser häufig auftretenden Standardsituation zu vereinfachen bietet die Standard-API die Schnittstelle [Iterator](#) an. Sie entbindet den Programmierer von der expliziten Ermittlung der Elementanzahl, sowie der indexgebundenen Traversierung.

Das nachfolgende Beispiel zeigt die Integration des Iterator-basierten Ansatzes für die bekannte Mengentraversierungsaufgabe

```

(1)import java.util.Iterator;
(2)import java.util.Vector;
(3)
(4)public class IteratorTest {
(5)    public static void main(String[] args) {
(6)        Vector v = new Vector();
(7)        v.add(new String("Berta"));
(8)        v.add(new String("Anna"));
(9)        v.add(new String("Cäsar"));
(10)
(11)        for (Iterator i = v.iterator() ; i.hasNext() ;) {
(12)            System.out.println(i.next());
(13)        } //for
(14)    } //main()
(15)} //class IteratorTest

```

Beispiel 11: Iterator-basierte Traversierung einer Objektmenge [IteratorTest.java](#)

Die Lösung enthebt den Programmierer zwar vom Aufwand die Elementanzahl explizit zu ermitteln und einem Speicherplatz zuzuweisen, jedoch wird mit dem Iterator-Objekt immernoch benannter Speicher (in Form der Variable `e`) angefordert, der ausschließlich der schleifeninternen Logik dient.

Erweiterte Schleifen-Syntax

Zur Behebung des Übelstandes der unnötigen expliziten Informationsermittlung im vorigen Beispiel existiert seit Java Version 1.5 eine abkürzenden Schreibweise für Mengentraversierungen.


```

(1)import java.util.Enumeration;
(2)import java.util.Vector;
(3)
(4)public class NewIteration {
(5)    public static void main(String[] args) {
(6)        Vector v = new Vector();
(7)        v.add(new String("Berta"));
(8)        v.add(new String("Anna"));
(9)        v.add(new String("Cäsar"));
(10)
(11)        for (Object o : v) {
(12)            System.out.println( o );
(13)        } //for
(14)    } //main()
(15)} //class NewIteration

```

Beispiel 12: Traversierung einer Objektmenge [NewIteration.java](#)

Das Beispiel zeigt die alternative Schreibweise, welche keine unnötigen, d.h. im Schleifenrumpf nicht benötigten, Daten ermittelt. Der Ausdruck innerhalb der for-Klammen typisiert die Elemente der Menge v als Object und weist sie temporär (konkret: für jeden Schleifendurchlauf einen Wert) der Variable o zu.

Die Entnahme aus der Objektmenge erfolgt unter Nutzung des allgemeinen Typs Object anstatt direkt auf den konkreten Typ String zurückzugreifen. Diese Asymmetrie erklärt sich aus Nutzung der Klasse Vector ohne Verwendung der in der Programmiersprache vorhandenen Generizitätsmechanismen. Nähere Ausführungen zu den damit verbundenen Möglichkeiten finden sich im Kapitel [Parametrische Polymorphie/Generics](#).

▲ 2.3.3 Ausnahmen und ihre Behandlung -- Exception Handling

Die Ausnahmebehandlung von Java ist konzeptionell und syntaktisch eng an das Pendant des ANSI-C++-Standards angelehnt. ([siehe Java Language Specification](#)).

Drei generelle Vorteile ergeben sich durch Verwendung von Ausnahmen:

- Separierung von Code und Fehlerbehandlung ([siehe Java Tutorial](#))
- Propagierung des Ausnahmezustandes; d.h. Behandlung muß nicht lokal in der fehlerauslösenden Routine erfolgen ([siehe Java Tutorial](#))
- Bildung von Fehlerklassen durch Gruppierung gleichartiger Ausnahmen ([siehe Java Tutorial](#))

Die generelle Syntax kann wie folgt beschrieben werden:

```

try {
    //statements which may throw an exception
    //... or create and throw an exception object manually
} catch (exceptionFoo e) {
    //handling of exception of type exceptionFoo
    //exception object is named e
} catch (exceptionBar e) {
    //handling of exception of type exceptionBar
    //exception object is named e
} catch (Exception e) {
    //handling all exceptions not handled by specialized handlers yet
    //exception object is named e
} finally {
    //executed regardless the execution state of the previous try block
} //finally

```

```

(1)public class ExceptionHandlingTest1 {
(2)    public static void main(String[] args) {
(3)        for (int i=2; i>=0; i--)
(4)            System.out.println(42/i);
(5)    } //main()
(6)} //class ExceptionHandlingTest1

```

Beispiel 13: Exception auslösender Code [ExceptionHandlingTest1.java](#)

Das Beispiel illustriert das Auftreten einer *java.lang.ArithmeticException*, verursacht durch die Division durch Null.

Alle in einen try-Block eingeschlossenen Anweisungen werden „überwacht“ ausgeführt. Das bedeutet, es erfolgt kein Programmabbruch beim Auftreten einer Ausnahme, sondern der direkte Anspruch eines *exception handlers*.

Exception Handler werden durch catch-Blöcke implementiert. Der Typ der zu behandelnden Exception wird als übergabeparameter angegeben. Das Laufzeitsystem sorgt für Auswahl der korrekten (d.h. zuständigen) Behandlungsroutine.

Alle Ausnahmen erben von der Klasse [Exception](#). Hierdurch kann auch ein Vorgabe-Exception-Handler installiert werden. Das folgende Beispiel zeigt diesen im Anschluß an die Behandlungsroutine der *ArithmeticException* Ausnahme. Eine Übersicht der in der Java-API definierten Ausnahme findet sich in: ([siehe Java API Specification](#))

Der Ausdruck `catch (Exception e)` übernimmt hierbei die Rolle des aus C++ bekannten `catch (...)`.

```
(1)public class ExceptionHandlingTest2 {
(2)    public static void main(String[] args) {
(3)        try {
(4)            for (int i=2; i>=0; i--)
(5)                System.out.println(42/i);
(6)        } catch (java.lang.ArithmeticException e) {
(7)            System.out.println("an arithmetic exception was thrown --
aborting program");
(8)            System.out.println("Exception's message "+e.getMessage() );
(9)            System.out.println("Stack trace: ");
(10)           e.printStackTrace();
(11)        }
(12)        catch (Exception e) {
(13)            System.out.println("an exception was thrown -- aborting
program");
(14)        } finally {
(15)            System.out.println("finished try block");
(16)        } //finally
(17)    } //main()
(18)} //ExceptionHandlingTest2
```

Beispiel 14: Ausnahmebehandlung [ExceptionHandlingTest2.java](#)

Das Programm liefert die Ausgabe:

```
$java ExceptionHandlingTest1
21
42
an arithmetic exception was thrown
finished try block
```

Der optional angebbare `finally`-Block wird immer ausgeführt, unabhängig davon ob der von `try` umschlossene Anweisungsteil erfolgreich (d.h. fehlerfrei) oder durch Exception (oder auch sonstige Sprungmechanismen wie `break`) verlassen wurde. Er bietet die Gelegenheit nach erfolgter Ausnahmebehandlung „Aufräumarbeiten“, wie das Schließen möglicherweise noch geöffneter Dateien, durchzuführen.

Nützliche Zusatzinformationen über die Art des aufgetretenen Ausnahmeereignisses können durch die Methoden `getMessage()` und `printStackTrace()` abgefragt werden. Beide Methoden sind von [java.lang.Throwable](#), der Superklasse von [java.lang.Exception](#), ererbt und stehen daher auf allen Ausnahmeobjekten zur Verfügung.

Ausnahmen die als Subklassen von [RuntimeException](#) realisiert sind müssen nicht explizit deklariert oder aufgefangen werden, da sie von [Instruktionen der virtuellen Maschine](#) erzeugt werden. Dies stellt einen Widerspruch zur erhobenen Forderung dar, daß alle Ausnahmen, die Subklassen von [Throwable](#) sind, explizit zu deklarieren und aufzufangen sind!

Eine korrekte Deklaration wäre jedoch unter praktischen Gesichtspunkten nicht praktikabel, da

beispielweise ein Fehler des Typs [InternalError](#) potentiell durch jede Methode erzeugt werden kann.

Abgesehen davon verhalten sich jedoch Laufzeit-Ausnahmen jedoch wie „normale“ Exceptions. Begründet durch Abwesenheit einer expliziten Deklaration sieht man auch -- außer in raren und begründeten Ausnahmefällen -- vom Auffangen und Behandeln durch den Programmierer ab.

Erzeugen eigener Ausnahmeereignisse

Neben durch das Laufzeitsystem generierten Ausnahmeereignissen können auch innerhalb des Programmcodes gezielt Exceptions durch den Anwender generiert werden. Hierfür existiert das [throw](#)-Statement.

```
(1)public class OwnException1 {
(2)    public static void main(String[] args) {
(3)        try {
(4)            System.out.println("throwing an arithmetic exception...");
(5)            throw new ArithmeticException();
(6)            //never gets here
(7)        } //try
(8)        catch (Exception e) {
(9)            System.out.println(e.toString() + " exception caught");
(10)        } //catch
(11)    } //main()
(12)} //class OwnException1
```

Beispiel 15: Anwenderdefiniert ausgelöste Ausnahme [OwnException1.java](#)

Das Codebeispiel zeigt das manuelle Erzeugen einer ArithmeticException.

Anmerkung: Nach throw angegebene Anweisungen können niemals erreicht werden, und führen bereits zum Übersetzungszeitpunkt zu einer entsprechenden Fehlermeldung.

Das bestehende vorgegebene System der Exceptions kann durch den Programmierer jederzeit um eigendefinierte Ausnahmen erweitert werden. Voraussetzung hierfür ist die Definition einer neuen Ausnahmeklasse; dies geschieht (im einfachsten Falle) durch Erben von [java.lang.Exception](#).

Das nachfolgende Beispiel illustriert dies:

```
(1)public class OwnException2 {
(2)    public static void main(String[] args) {
(3)        try {
(4)            System.out.println("throwing myException...");
(5)            throw new myException();
(6)        } catch (Exception e) {
(7)            System.out.println(e.toString() + " exception caught");
(8)        } //catch
(9)    } //main()
(10)} //class OwnException2
(11)
(12)class myException extends Exception {
(13)} //class myException
```

Beispiel 16: Eigendefinierte Ausnahme [OwnException2.java](#)

Für alle Ausnahmen die nicht lokal behandelt werden, kann durch Angabe der [throws-Liste](#) die Ausnahmenbehandlung an den Aufrufer weitergereicht werden.

```
(1)public class OwnException3 {
(2)    public static void main(String[] args) {
(3)        try {
(4)            exceptionProne();
(5)        } catch (myException myE) {
(6)            System.out.println("Exception of type myException caught!");
(7)        } //catch
(8)    } //main()
(9)
(10)    public static void exceptionProne() throws myException {
(11)        throw new myException();
(12)    } //exceptionProne()
(13)} //class OwnException3
(14)
(15)class myException extends Exception {
```

```
(16)} //class myException
```

Beispiel 17: Propagierung der Ausnahmebehandlung [OwnException3.java](#)

Im Beispiel wird die eigendefinierte Ausnahme `myException` nicht innerhalb der Methode `exceptionProne` behandelt, sondern an den Aufrufer -- in diesem Beispiel die `main`-Methode -- zur Behandlung weitergereicht.

Der Übersetzer prüft hier das Vorhandensein eines `try-catch`-Blockes innerhalb von `main` ab. Das folgende Beispiel zeigt eine Erweiterung des vorhergehenden, dergestalt, daß auch die `main`-Methode mit einem `throws` versehen ist. In diesem Fall wird das Ausnahmeereignis an das Laufzeitsystem weitergereicht; welches in der Konsequenz die Ausführung terminiert.

Generell gilt in Java die *catch or throw*-Regel, die besagt, daß ein Ausnahmeereignis entweder aufgefangen und behandelt (*catch*) oder weitergereicht (*throw*) werden muß.

```
(1)public class OwnException4 {
(2)    public static void main(String[] args) throws myException    {
(3)        throw new myException();
(4)    } //main()
(5)} //class OwnException4
(6)
(7)class myException extends Exception {
(8)} //class myException
```

Beispiel 18: Propagierung eines Ausnahmeereignisses an das Laufzeitsystem [OwnException4.java](#)

Auch die Kombination der beiden vorgestellten Ansätze ist möglich ...

Im abschließenden Codebeispiel wird neben der lokalen Ausnahmebehandlung (`catch`-Block innerhalb `exceptionProne()`) auch eine Behandlung innerhalb der aufrufenden Methode (`main`) durchgeführt. Hierzu wird das aufgefangene Ausnahmeereignis innerhalb der Behandlungsroutine erneut mittels `throw` ausgelöst.

```
(1)public class OwnException5 {
(2)    public static void main(String[] args) {
(3)        try {
(4)            exceptionProne();
(5)        } catch (myException e) {
(6)            System.out.println("exception myException caught within
main method");
(7)        } //catch
(8)    } //main()
(9)
(10)    public static void exceptionProne() throws myException {
(11)        try {
(12)            throw new myException();
(13)        } catch (myException e) {
(14)            System.out.println("exception myException caught within
method exceptionProne");
(15)            System.out.println("re-throwing...");
(16)            throw e;
(17)        } //catch
(18)    } //exceptionProne()
(19)} //class OwnException5
(20)
(21)class myException extends Exception {
(22)} //class myException
```

Beispiel 19: Behandlung und Weiterreichung einer Ausnahme [OwnException5.java](#)

▲ Zwangsbedingungen

Zur Steigerung der Qualität des entstehenden Systems gestattet Java die Definition von *Zwangsbedingungen* (*Assertion*), deren Einhaltung während der Ausführung geprüft werden kann. Ist eine solche Bedingung nicht erfüllt, so erfolgt ein Programmabbruch.

Im Unterschied zur Möglichkeit durch Selektionsausdrücke die Rückgabewerte von Methodenaufrufen auszuwerten oder durch Ausnahmebehandlung gezielt und benutzerdefiniert auf Fehlersituationen zu reagieren bieten die Zwangsbedingungen sowohl eine gegenüber den beiden genannten Ansätzen signifikant kompaktifizierte Syntax an, die gleichzeitig weniger Freiheitsgrade in der Behandlung der Fehlersituation bietet.

Zusätzlich kann die Prüfung von Zwangsbedingungen zur Laufzeit statisch durch einen Schalter der Ausführungsumgebung gesteuert werden. Auf dieser Basis eignen sie sich gut für die Formulierung verschiedenster Konsistenzprüfungen, die später im Produktivbetrieb zur Steigerung der Ausführungsgeschwindigkeit deaktiviert werden können.

Die allgemeine Syntax einer Zwangsbedingung lautet:

```
assert Boole'scher-Ausdruck (: Ausdruck)opt
```

Generell werden Zwangsbedingungen durch das Schlüsselwort `assert` eingeleitet. Darauf folgt ein Boole'scher Ausdruck, der erfüllt sein muß. Liefert die Auswertung dieses Ausdrucks den Wahrheitswert `false`, so erfolgt der Programmabbruch. Ist zusätzlich, nach dem separierenden Doppelpunkt, ein Ausdruck angegeben, so wird dieser zur Konstruktion eines [AssertionError](#)-Objekts herangezogen.

Aus den zugelassenen Parametertypen eines solchen Objekts ergeben sich die möglichen angebbaren Ausdruckstypen als: [boolean](#), [char](#), [double](#), [float](#), [int](#), [long](#) und [Object](#).

Das nachfolgende Beispiel zeigt den Einsatz einer einfachen Zwangsbedingung, deren Fehlschlag ohne anwenderdefinierte Konstruktion `AssertionError`-Objekts behandelt wird:

```
(1)public class AssTest1 {
(2)    public static void main(String[] args) {
(3)        Object o = null;
(4)        //...
(5)        assert o != null;
(6)    } //main()
(7)} //class AssTest1
```

Beispiel 20: Einfache Zwangsbedingung [AssTest1.java](#)

Das Beispiel deklariert eine Variable `o` als Ausprägung der Klasse `Object` und initialisiert diese mit `null`. Im späteren Verlauf der Ausführung soll geprüft werden, ob zwischenzeitlich eine Initialisierung erfolgt ist. Ist dies nicht der Fall, so ist eine Programmfurthersetzung nicht sinnvoll. Diese Prüfung, verbunden mit der impliziten Forderung den Ablauf bei ihrer Nichterfüllung zu terminieren, ist als Zwangsbedingung realisiert.

Zur Verwendung der Zwangsbedingungen ist die Übersetzung mindestens konform zur Sprachversion 1.4 notwendig. Aktiviert wird diese Quellcodeinterpretation durch Übergabe des Wertes „1.4“, oder höher, als Parameter des Übersetzerschalters `-source`. Wird ein Wert kleiner als 1.4 gewählt, so wird das Schlüsselwort `assert` nicht als solches interpretiert, sondern kann als Identifier zur Benennung von Klassen, Attributen oder Methoden benutzt werden.

Im selben Sinne ist es ebenso zwingend erforderlich die Prüfung von Zwangsbedingung innerhalb der Laufzeitumgebung durch den Schalter `-enableassertions` zu aktivieren. Andernfalls werden alle `assert`-Anweisungen ungeprüft ignoriert. Aus dieser Forderung ergibt sich, daß mindestens Version 1.4 der Laufzeitumgebung benötigt wird um die Interpretation von Zwangsbedingungen zu aktivieren.

Die Ausführung des Beispiels liefert daher, bei aktiviertem Schalter die Ausgabe:

```
Exception in thread "main" java.lang.AssertionError
    at AssTest1.main(AssTest1.java:5)
```

Zur Einschränkung der Zwangsbedingungsauwertung gestattet die Laufzeitumgebung die Spezifikation derjenigen Klassen, für die diese Bedingungen ausgewertet werden sollen. Hierzu werden nach dem Schlüsselwort `enableassertions`, abgetrennt durch einen Doppelpunkt, die Namen der zu berücksichtigenden Klassen angegeben werden. Für das obenstehende Beispiel sind daher die Aufrufe `java -enableassertions AssTest1` und `java -enableassertions:AssTest1 AssTest1` äquivalent.

Ebenfalls durch Kommandozeilenschalter kann die (De-)Aktivierung der Prüfung der in den Standard-API-Klassen formulierten Zwangsbedingungen gesteuert werden. Hierfür stehen die

Schalter `-enableSystemAssertions` bzw. `-disableSystemAssertions`. Die Möglichkeiten der klassengenauen Steuerung stehen jedoch in diesem Falle nicht zur Verfügung.

Die Möglichkeit der Übergabe von Parametern an das automatisiert durch das Laufzeitsystem erzeugte `AssertionError`-Objekt bietet sich insbesondere zur Dokumentation der Abbruchursache an. So zeigt das nachfolgende Beispiel prüft durch Aufruf der Methode `exists` der Klasse `File` ob die Datei `test` im aktuellen Dateisystemkatalog angelegt ist. Ist dies nicht erfüllt, so wird eine festgelegte Zeichenkette dem Konstruktor von `AssertionError` übergeben. Die Zeichenkette kann, als Ausprägung der Klasse `String` zur Konstruktion eines Fehlerobjektes verwendet werden, da sie gemäß der Typrestriktion eine gültige Instanz von `Object` repräsentiert. Gleichzeitig kann die Methode `exists` als Ausdruck nach dem `assert`-Schlüsselwort angegeben werden, da die Ausführung der Methode einen Rückgabewert vom Typ `boolean` liefert und damit der Gesamtausdruck als Wahrheitswert ausgewertet werden kann.

```
(1)import java.io.File;
(2)
(3)public class AssTest2 {
(4)    public static void main(String[] args) {
(5)        //...
(6)        assert (new File("test")).exists() : "File 'test' does not exist";
(7)    } //main()
(8)} //class AssTest2
```

Beispiel 21: Zwangsbedingung mit anwenderdefinierter Fehlerobjekt [AssTest2.java](#)

Zwar sollte nach Nichterfüllung einer Zwangsbedingung die Applikationsausführung beendet werden, wie es auch im Standardfalle durch das Laufzeitsystem geschieht. Jedoch kann dieses Verhalten mit Mitteln des Ausnahmebehandlung unterbunden werden. Hierzu muß eine Zwangsbedingungsauswertung in einen `try`-Block einbettet werden. Findet sich im zugeordneten `catch`-Bereich eine Klausel für den `AssertionError` so wird diese ausgeführt und anschließend die Programmverarbeitung normal fortgesetzt. Das nachfolgende Beispiel zeigt diese Anwendung.

```
(1)import java.io.File;
(2)
(3)public class AssTest3 {
(4)    public static void main(String[] args) {
(5)        //...
(6)        try {
(7)            assert false : "this assertion fails definitely";
(8)        } catch (AssertionError ae) {
(9)            System.out.println("caught assertion error");
(10)        } //catch
(11)        System.out.println("continuing after error ");
(12)    } //main()
(13)} //class AssTest3
```

Beispiel 22: Abfangen einer fehlschlagenden Zwangsbedingung [AssTest3.java](#)

Das im Beispiel gezeigte Verhalten ist zwar syntaktisch korrekt und wird auch im angestrebten Sinne ausgeführt, sollte jedoch nur mit Bedacht angewandt werden, da es die Semantik einer Zwangsbedingung aufweicht.

Insgesamt empfiehlt sich die Verwendung von Zwangsbedingungen lediglich für eine engumrissene Klasse von Fehlern, wie Nachbedingungen von Methoden, die einen internen Systemzustand dokumentieren, der durch keine Modifikation in einen konsistenten Zustand überführt werden könnte, er eine Ausführungsfortsetzung sinnvoll erscheinen läßt. Insbesondere solche, die durch Anwenderinteraktion begründet sind (wie fehlerhafte Parameterübergabe, etc.) sollten durch Ausnahmen behandelt werden.

▲ 2.4 Von komplexen zu objektorientierten Datenstrukturen

▲ 2.4.1 Arrays

Wie fast alle prozeduralen Programmiersprachen unterstützt auch Java die Zusammenfassung

beliebiger gleichartiger Elemente zu geordneten, nicht zwingend duplikatfreien, Mengen; die sog. *Feldern, Arrays oder Vektoren*.

Arrays in Java sind nicht Bestandteil des [built-in Typsystems](#), sondern bereits als [Klasse innerhalb der Java API](#) realisiert.

Wie in anderen Programmiersprachen üblich stellen Javaarrays einen zusammenhängen kontinuierlichen Speicherbereich dar.

Bereits innerhalb der Standardsignatur der main-Methode wird ein Array verwendet:

```
public static void main(String[] args).
```

Einige Charakteristika:

- Statische Größenfestlegung.
Arraygrenzen können nach Definition nicht mehr verändert werden.
Vorgriff: Dies ist mit der API Klasse [Vector](#) möglich.
- Arrays sind (*first class*) Objekte.
=> Sie müssen durch den Programmierer explizit mit new erzeugt werden.
- Die Inhaltselemente sind immer beginnend mit 0 nummeriert.
- Zugriffe über die Arraygrenzen hinaus werden durch die `ArrayIndexOutOfBoundsException` abgefangen.
- Definition anonymer (d.h. namenloser) Arrays möglich.
- Definition „mehrdimensionaler“ Arrays möglich.
...dazu später mehr.
- Array können bereits während der Definition initialisiert werden.
Hierdurch können auch dynamisch allokierte Arrays realisiert werden.
- Zeichenketten (Strings) sind *nicht* als *Array of Character*, sondern als [eigene API-Klasse](#), realisiert.
- Der Zugriff auf einzelne Arrayelemente erfolgt über Index (=berechenbare Adresse).
`a[i]` greift auf das i+1-te Element des Arrays a zu.
- Arraykomponenten werden bei der Erzeugung mit new initialisiert.
Im Falle primitiver Datentypen wird als Vorgabewert 0 für alle numerischen Datentypen, false für boolean und Unicode `\u0000` für char gesetzt. Objektwertige Typen werden durch ihren Konstruktor initialisiert.

Syntax:

ArrayCreationExpression:

```
new PrimitiveType DimExprs DimsOptional
new TypeName DimExprs DimsOptional
new PrimitiveType DimsOptional ArrayInitializer
new TypeName Dims ArrayInitializer
```

DimExprs:

```
DimExpr
DimExprs DimExpr
```

DimExpr:

```
[ Expression ]
```

Dims:

```
[ ]
Dims [ ]
```

```
(1)public class Array1 {
(2)    public static void main(String[] args) {
(3)        int                firstArray[]    = new int[10];
(4)        int[]              secondArray     = new int[10];
(5)        double[]           thirdArray      = {3.14, 2+5, 42.0};
(6)
(7)        System.out.println("length of firstArray="+ firstArray.length );
(8)        System.out.println("length of secondArray="+ secondArray.length );
(9)        System.out.println("length or thirdArray="+ thirdArray.length );
(10)       for (int i=0; i<thirdArray.length; i++)
(11)           System.out.println("thirdArray["+i+"]="+thirdArray[i]);
(12)
(13)       System.out.println("lenght of anonymous array="+ (new byte[]
{1,2,3}).length);
(14)    } //main()
(15)} //class Array1
```


Beispiel 23: Arrayerzeugung und Initialisierung [Array1.java](#)

Der Code aus Beispiel 16 definiert zunächst zwei `int`-Array der Größe 10. Die beiden Syntaxvarianten sind (wie aus C/C++ bekannt) äquivalent. `thirdArray` wird bereits während der Definition mit `double`-Werten initialisiert. Die Größe muß nicht explizit angegeben werden, die errechnet sich automatisch aus der Anzahl der angegebenen Ausdrücke.

Die letzte Definition eines Arrays im Beispiel erfolgt anonym. Der so erzeugte Array steht nach Verlassen des `System.out.println`-Ausdrucks nicht mehr zur Verfügung.

Arrays deren Elemente eigendefinierte Typen sind werden entsprechend mit *eigenerTyp*[] *arrayName* ... definiert.

Mehrdimensionale Arrays ...

werden nicht explizit unterstützt, sondern als Arrays von Arrays behandelt.

```
(1)public class Array2 {
(2)    public static void main(String[] args) {
(3)        int[][] multiDimensional = new int[2][3];
(4)
(5)        int dimR = multiDimensional.length;
(6)        int dimC = multiDimensional[0].length;
(7)
(8)        System.out.println("Lenght of multiDimensional="+ dimR);
(9)        System.out.println("Lenght of multiDimensional[0]="+dimC );
(10)
(11)        for(int i=0; i<dimR; i++)
(12)            for(int j=0; j<dimC; j++)
(13)                multiDimensional[i][j] = i*dimC + j+1;
(14)
(15)        System.out.println("Content of multiDimensional:");
(16)        for(int i=0; i<dimR; i++) {
(17)            for(int j=0; j<dimC; j++) {
(18)                System.out.print(multiDimensional[i][j]+ " ");
(19)            } //for
(20)            System.out.println();
(21)        } //for
(22)    } //main()
(23)} //class Array2
```

Beispiel 24: Mehrdimensionale Arrays [Array2.java](#)**BildschirmAusgabe:**

```
$java Array2
Lenght of multiDimensional=2
Lenght of multiDimensional[0]=3
Content of multiDimensional:
1 2 3
4 5 6
```

Die Angabe mehrerer Dimensionsgrößen bei der Definition stellt eine Kurzform für die verschachtelte Variante dar. So ist das nachfolgende Codefragment äquivalent zum vorhergehenden Beispiel:

```
(1)public class Array3 {
(2)    public static void main(String[] args) {
(3)        int[][] multiDimensional = new int[2][];
(4)        for (int i=0; i<multiDimensional.length; i++)
(5)            multiDimensional[i] = new int[3];
(6)
(7)        int dimR = multiDimensional.length;
(8)        int dimC = multiDimensional[0].length;
(9)
(10)        System.out.println("Lenght of multiDimensional="+ dimR);
(11)        System.out.println("Lenght of multiDimensional[0]="+dimC );
(12)
(13)        for(int i=0; i<dimR; i++)
(14)            for(int j=0; j<dimC; j++)
(15)                multiDimensional[i][j] = i*dimC + j+1;
```



```

(16)
(17)         System.out.println("Content of multiDimensional:");
(18)         for(int i=0; i<dimR; i++) {
(19)             for(int j=0; j<dimC; j++) {
(20)                 System.out.print(multiDimensional[i][j]+ " ");
(21)             } //for
(22)             System.out.println();
(23)         } //for
(24)     } //main()
(25)} //class Array3

```

Beispiel 25: verschachtelte mehrdimensionale Arraydefinition [Array3.java](#)

Die Definition

```
int[][] multiDimensional = { {1,2,3}, {100,200,300} };
```

Definiert einen Array der zwei Array, jeweils der Länge 3, enthält, und initialisiert diese mit den angegebenen Werten.

Unterscheiden sich die beiden implizit spezifizierten Arrays in der Größe, so definiert die Anzahl des ersten angegebenen Arrays die Elementzahl (Zeilenlänge) für alle folgenden (Zeileneinträge).

Duplizieren von Arrays:

Für die häufig benötigte Anwendung den vollständigen Inhalt eines Arrays in einen zweiten Array gleicher Dimension(en) zu kopieren existiert die clone-Methode.

```

(1)public class Array4 {
(2)    public static void main(String[] args) {
(3)        int[] ia = {1,2,3,4,5};
(4)        int[] ib = new int[ia.length];
(5)
(6)        System.out.println("ia==ib = "+ (ia==ib));
(7)
(8)        System.out.println("content of ia:");
(9)        for(int i=0; i<ia.length; i++)
(10)            System.out.print(ia[i]+" ");
(11)        System.out.println("");
(12)
(13)        System.out.println("content of ib:");
(14)        for(int i=0; i<ib.length; i++)
(15)            System.out.print(ib[i]+" ");
(16)        System.out.println("");
(17)
(18)
(19)        ib = (int[]) ia.clone();
(20)        System.out.println("content of ib after cloning:");
(21)        for(int i=0; i<ib.length; i++)
(22)            System.out.print(ib[i]+" ");
(23)        System.out.println("");
(24)
(25)        System.out.println("ia==ib = "+ (ia==ib));
(26)
(27)        System.out.println("Setting ia=ib...");
(28)        ia=ib;
(29)        System.out.println("ia==ib = "+ (ia==ib));
(30)    } //main()
(31)} //class Array4

```

Beispiel 26: Duplizierung eines Arrayinhaltes mit der clone-Methode [Array4.java](#)

BildschirmAusgabe:

```

$java Array4
ia==ib = false
content of ia:
1 2 3 4 5
content of ib:
0 0 0 0 0
content of ib after cloning:
1 2 3 4 5

```

```
ia==ib = false
```

Das Beispiel definiert zunächst den Array `ia` per expliziter Initialisierung. Ein zweiter Array, `ib` wird mit derselben Länge wie `ia` definiert.

Wie zu erwarten sind die Arrayobjekte verschieden, d.h. sie belegen unterschiedliche Speicherplätze.

Die `clone`-Methode dupliziert den Inhalt des Arrays `ia` und weist in `ib` zu.

In den letzten Zeilen wird der Variable `ia` der Wert von `ib` zugewiesen. Mithin der Array `ia` durch `ib` überschrieben. Das ursprüngliche `ia` kann damit nicht mehr durch den Programmierer referenziert werden, und wird für den Garbage Collector als freizugeben markiert.

Die **`ArrayStoreException`** wird ausgeworfen, sobald ein Typkonflikt beim Einfügen eines Arrayelementes zur Laufzeit auftritt; statisch erkennbare Typkonflikte werden bereits durch den Übersetzer gemeldet.

```
(1)public class Array5 {
(2)    public static void main(String[] args) {
(3)        Test2[] ta = new Test2[5];
(4)        Test1[] tb = ta;
(5)
(6)        try {
(7)            tb[0] = new Test1();
(8)        } //try
(9)        catch (ArrayStoreException e) {
(10)            System.out.println(e);
(11)        } //catch
(12)    } //main()
(13)} //class Array5
(14)
(15)class Test1 {
(16)} //class Test1
(17)
(18)class Test2 extends Test1 {
(19)} //class Test2
```

Beispiel 27: Typkonflikt beim Einfügen, der zur Auslösung einer `ArrayStoreException` führt [Array5.java](#)

Obwohl der Typ der Arraykomponenten von `tb` als `Test1` deklariert war, führt die Zuweisung innerhalb des `try`-Blockes zu einer `ArrayStoreException`.

Ursache: Durch die Initialisierung mit Elementen des Typs `Test2`, der eine Spezialisierung von `Test1` bildet, wird auch der erwartete Inhaltstyp von `Test1` verändert (konkret: spezialisiert).

▲ 2.4.2 Klassen

Klassen und Objekte stellen das namensgebende Hauptabstraktionsmittel in der objektorientierten Programmierung dar.

Prinzipiell lassen sich aus Sicht der OO-Programmierung drei Arten von Sprachen unterscheiden:

- Klassenlose Sprachen, wie C, Pascal, etc.
- Klassenbasierte hybride Sprachen, die Objekten und Vererbung unterstützen -- jedoch ihre Verwendung nicht auf allen Ebenen durchgängig erzwingen, wie C++ und Java
- (rein) objektorientierte Sprachen, wie SmallTalk, Eiffel, etc.

Gemäß dieser Einordnung ist Java als hybride objektorientierte Sprache anzusehen. Dies rührt hauptsächlich von der Umsetzung des [build-in Typsystems](#) als einfache Werte -- anstatt first class objects -- her.

In der Praxis zieht dies jedoch zumeist keine allzugrossen Einschränkungen nach sich.

Die Verwendung von Klassen ist in Java, im scharfen Gegensatz zu C++, zwingend. So ist es nicht möglich ein Programm ohne zumindest eine (Haupt-)Klasse zu schreiben.

Jede Klasse bildet einen abgeschlossenen Sichtbarkeitsbereich (auch: Scope) für die darin definierten [Attribute](#) und [Methoden](#).

Klassendeklaration

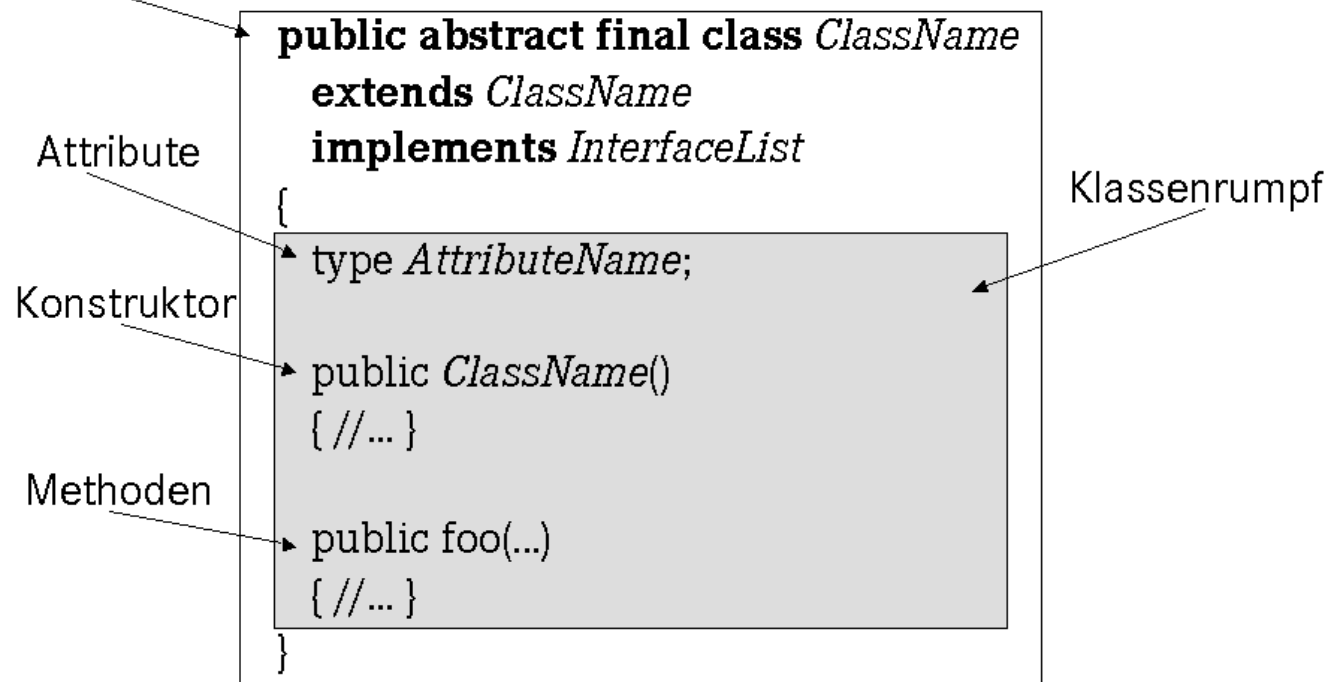


Abbildung 7: Bestandteile einer Klasse (click on image to enlarge!)

Eine Java-Klasse besteht aus den in der Abbildung dargestellten Teilen:

Klassendeklaration: Sie benennt die Klasse eindeutig, und legt die allgemeinen Charakteristika fest. Das Aussehen der Deklaration kann wie folgt beschrieben werden:

Syntaxelement	Semantik
<code>public</code>	Die Klasse ist allgemein sichtbar. Vorgabegemäß kann eine Klasse nur von Klassen desselben Packages genutzt werden. Durch das Schlüsselwort <code>public</code> wird sie auch außerhalb des umgebenden Packages sichtbar und zugreifbar. Die namensgebende Klasse einer Java-Quellcode-Datei, die auch die <code>main</code>-Methode enthalten kann, muß zwingend als <code>public</code> erklärt sein.
<code>abstract</code>	es können keine Objekte dieser Klasse erzeugt werden. Abstrakte Klassen dienen zur Strukturierung des Entwurfs, um gemeinsame Merkmale verschiedener Klassen zentralisiert ausdrücken zu können.
<code>final</code>	von dieser Klasse kann nicht geerbt werden. Die Entscheidung eine Klasse als <code>final</code> zu deklarieren ist designgetrieben. In der Verwendung der Klasse ergeben sich keine Unterschiede. Jedoch können so Vererbungsäste als abgeschlossen gekennzeichnet werden, um auszudrücken, daß an dieser Stelle keine Weiterentwicklung erfolgen soll. Das Schlüsselwort verhindert einen entsprechenden Versuch durch einen Fehler zum Übersetzungszeitpunkt. Beispiele aus der Java-API: java.io.FileDescriptor, die Wrappertypen: Boolean, Integer, etc.
<code>class <i>ClassName</i></code>	Name der Klasse
<code>extends <i>ClassName</i></code>	Die Klasse erbt von der Klasse <i>ClassName</i> In Java ist nur einfache Vererbung zugelassen, daher kann an dieser Stelle auch maximal eine Superklasse spezifiziert werden.
<code>implements <i>InterfaceList</i></code>	Die Klasse implementiert die in der <i>InterfaceList</i> aufgeführten Schnittstellen. Die Schnittstellennamen werden durch Kommata voneinander abgetrennt.
<code>{</code> <code> <i>ClassBody</i></code> <code>}</code>	Ausprogrammierter Klassenrumpf.

Ein umfangreicheres Beispiel:

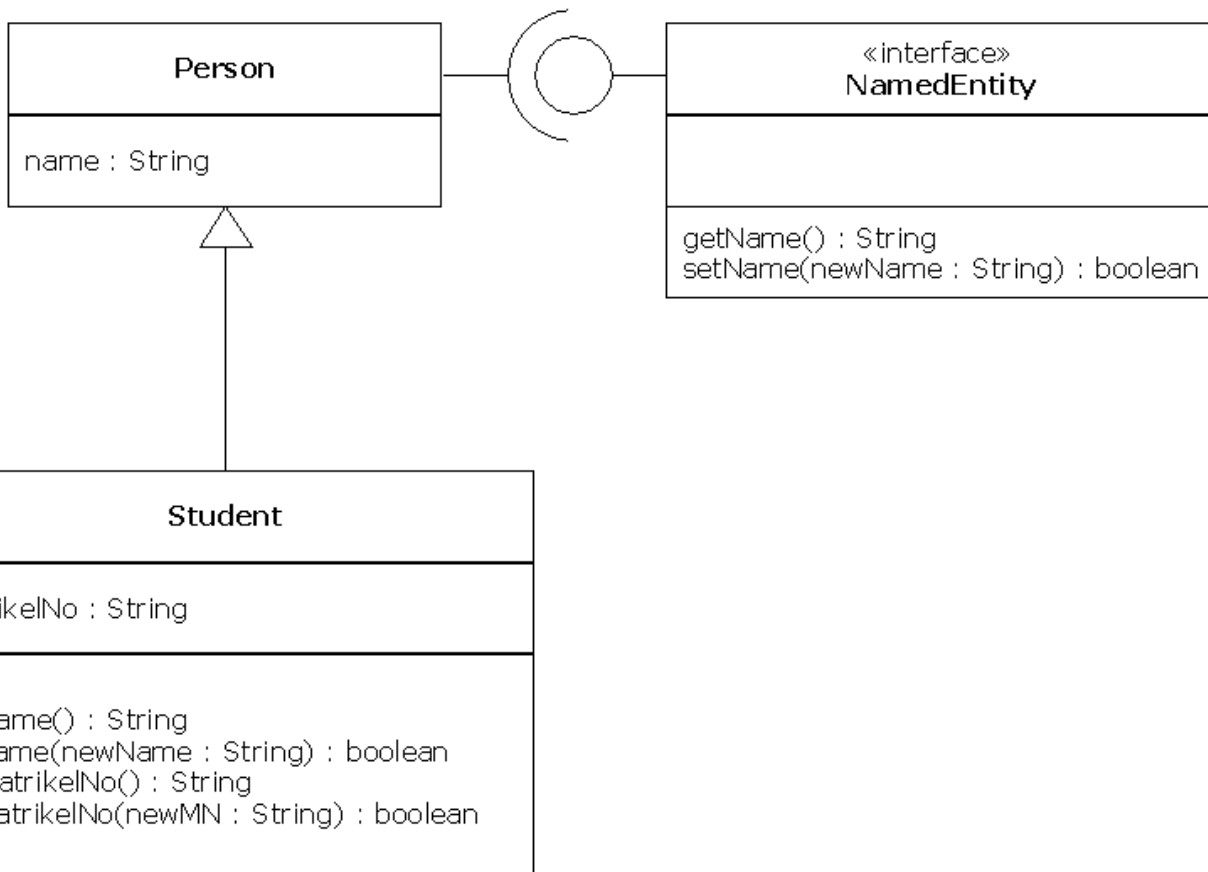


Abbildung 8: UML-Klassendiagramm (click on image to enlarge!)

```

(1) public class Student extends Person implements NamedEntity {
(2)     private String matrikelNo;
(3)
(4)     public Student(String matrikelNo) {
(5)         this.matrikelNo = matrikelNo;
(6)     } //constructor
(7)
(8)     public String getMatrikelNo() {
(9)         return matrikelNo;
(10)    } //getMatrikelNo;
(11)
(12)    public boolean setMatrikelNo(String matrikelNo) {
(13)        if (matrikelNo.compareTo("")==0) {
(14)            this.matrikelNo = matrikelNo;
(15)            return true;
(16)        } else
(17)            return false;
(18)    } //setMatrikelNo()
(19)
(20)    public String getName() {
(21)        return name;
(22)    } //getName()
(23)
(24)    public boolean setName(String newName) {
(25)        if (newName.compareTo("")!=0) {
(26)            name = newName;
(27)            return true;
(28)        } else
(29)            return false;
(30)    } //setName()
(31)
(32)    public String toString() {
(33)        return ("Name: "+this.getName()+"\nMatrikelnummer: "+this.
getMatrikelNo() );
(34)    } //toString()
(35)
  
```

```

(36)    public static void main(String[] args) {
(37)        Student mario = new Student("0793022");
(38)        System.out.println( mario.getMatrikelNo() );
(39)        System.out.println("mario.toString() returns:\n"+mario.toString() );
(40)        mario.setName("Mario Jeckle");
(41)        System.out.println("mario.toString() returns:\n"+mario.toString() );
(42)    } //main()
(43)} //class Student
(44)
(45)class Person {
(46)    String name;
(47)} //class Person
(48)
(49)interface NamedEntity {
(50)    String getName();
(51)    boolean setName(String newName);
(52)} //interface NamedEntity

```

Beispiel 28: Beispiel einer ausprogrammierten Klasse [Student.java](#)

Bildschirmausgabe:

```

0793022
mario.toString() returns:
Name: null
Matrikelnummer: 0793022
mario.toString() returns:
Name: Mario Jeckle
Matrikelnummer: 0793022

```

Innere Klassen

Seit Java v1.1 besteht auch die Möglichkeit Klassen innerhalb von Klassen zu definieren.

Merkmale:

- Objekte der inneren Klasse können auf private Daten des erstellenden Objektes zugreifen.
- Innere Klassen unterliegen dem Sichtbarkeitsbereich der umgebenden Klasse, d.h. sie sind vor Zugriffen anderer Klassen geschützt.
- Stark ereignisbezogene Anwendungen wie GUI-Bibliotheken nutzen diese Definitionsvariante sehr stark aus.
- Für inner classes wird eine eigenständige class-Datei mit der Namenskonvention *UmgebendeKlasse \$innereKlasse* erzeugt.

Innere Klassen bilden daher kein originäres Sprachmerkmal, sondern werden durch den Compiler in „normale“ Klassen umgesetzt; für die virtuelle Maschine ist diese Mimik (nahezu) transparent.

(Naive) Erweiterung des vorhergehenden Beispiels: Die Matrikelnummer ist als eigenständige Klasse innerhalb von Student2 realisiert:

```

(1)public class Student2 extends Person implements NamedEntity {
(2)    public Student2(String newMatrikelNo) {
(3)        matrikelNo = new MatrikelNo(newMatrikelNo);
(4)    } //constructor
(5)
(6)    class MatrikelNo {
(7)        private String matrikelNo;
(8)
(9)        public MatrikelNo(String newMatrikelNo) {
(10)            matrikelNo = newMatrikelNo;
(11)            if (this.checkMatrikelNo() != true)
(12)                System.out.println("illegal MatrikelNo");
(13)        } //constructor
(14)        public boolean checkMatrikelNo() {
(15)            return( this.matrikelNo.length() == 7 ? true : false);
(16)        } //end checkMatrikelNo()
(17)
(18)        public String getMatrikelNo() {
(19)            return matrikelNo;
(20)        } //getMatrikelNo()
(21)

```

```

(22)         public boolean setMatrikelNo(String newMatrikelNo) {
(23)             matrikelNo = newMatrikelNo;
(24)             if (this.checkMatrikelNo() == true)
(25)                 return true;
(26)             else
(27)                 return false;
(28)         } //setMatrikelNo()
(29)     } //MatrikelNo;
(30)
(31)
(32)     private MatrikelNo matrikelNo;
(33)
(34)     public String getMatrikelNo() {
(35)         return matrikelNo.getMatrikelNo();
(36)     } //getMatrikelNo;
(37)
(38)     public boolean setMatrikelNo(String matrikelNo) {
(39)         return true;
(40)     } //setMatrikelNo()
(41)
(42)     public String getName() {
(43)         return name;
(44)     } //getName()
(45)
(46)     public boolean setName(String newName) {
(47)         if (newName.compareTo("") != 0) {
(48)             name = newName;
(49)             return true;
(50)         } else
(51)             return false;
(52)     } //setName()
(53)
(54)     public String toString() {
(55)         return ("Name: " + this.getName() + "\nMatrikelnummer: " + this.
getMatrikelNo() );
(56)     } //toString()
(57)
(58)     public static void main(String[] args) {
(59)         Student2 mario = new Student2("0793022");
(60)         System.out.println( mario.getMatrikelNo() );
(61)
(62)         Student2 hans = new Student2("1234567X");
(63)     } //main()
(64)} //class Student2
(65)
(66)class Person {
(67)    String name;
(68)} //class Person
(69)
(70)interface NamedEntity {
(71)    String getName();
(72)    boolean setName(String newName);
(73)} //interface NamedEntity

```

Beispiel 29: Realisierung der Matrikelnummer als innere Klasse [Student2.java](#)

Anonyme Klassen:

Als weitere Besonderheit besteht die Möglichkeit die eingebettete Klasse anonym zu definieren. ([siehe Java Language Specification](#))

Syntaktisch folgt die gesamte Klassendefinition auf eine mit `new` eingeleitete Objekterzeugung. Unbenannte Klassen verfügen über keinen Konstruktor, da dieser konsequenterweise auch namenlos sein müsste.

Syntax:

```

new BaseClass (Parameters) {
    //inner class methods and data
};

```

Hinweis: Man beachte das (zwingende) Semikolon am Ende des Ausdrucks!

```
(1)public class Anonymous {
(2)    private TestClass test() {
(3)        return new TestClass() {
(4)            public void hello() {
(5)                System.out.println("overridden test!");
(6)            } //hello()
(7)        }; //class TestClass
(8)    } //test()
(9)
(10)   public static void main(String[] args) {
(11)       TestClass myRV;
(12)       myRV = (new Anonymous()).test();
(13)       myRV.hello();
(14)
(15)       System.out.println( myRV instanceof TestClass);
(16)       System.out.println( myRV.getClass().getName() );
(17)   } //end main()
(18)} //class Anonymous
(19)
(20)class TestClass {
(21)    public void hello() {
(22)        System.out.println("original");
(23)    } //hello()
(24)} //class TestClass
```

Beispiel 30: Anonyme innere Klasse [Anonymous.java](#)

BildschirmAusgabe:

```
$java Anonymous
overridden test!
true
Anonymous$1
```

In der durch die Methode `test` retournierten Ausprägung der Klasse `TestClass` ist die Methode `hello` überschrieben. Da jede innere anonyme Klasse eine bestehende Klasse aus Ausgangsbasis besitzen muß kann die zurückgegebene Ausprägung einer Speicherzelle dieses Typs zugewiesen werden. Entsprechend liefert der [instanceof-Operator](#) auch `true` für den Typtest zurück.

Durch die Redefinition der Methode `hello` wird innerhalb von `main` nicht die ursprüngliche, sondern die überschreibende der anonymen Klasse aufgerufen.

Die letzte Programmzeile in `main` bildet einen Vorgriff auf die noch zu behandelnde [Reflection API](#), welche die Gewinnung von Modellinformationen zur Laufzeit erlaubt. Konkret ermittelt die abgebildete Zeile den Namen der Klasse des in `myRV` gespeicherten Objekts. Java setzt für innere Klassen einen Namen aus der umgebenden Klasse und dem Namen der inneren Klasse, abgetrennt durch das Dollarsymbol, zusammen. Anonyme Klassen werden aufsteigend nummeriert. Daher im Beispiel der Surrogatname `Anonymous$1` für die erste anonyme Klasse innerhalb der Klasse `anonymous`.

Im Grunde genommen handelt es sich bei anonymen inneren Klassen de facto um eine besondere Art der [Vererbung](#), welche bereits bekannte Methoden überschreibt. Das überladen existierender Methoden, sowie die Erweiterung der Superklasse um zusätzliche Attribute oder Methoden ist nicht möglich.

Vorwärtsverweis: Ein [reales Beispiel](#) für die Nutzung dieses Sprachmechanismus wird in Kapitel 3 vorgestellt.

Im Kontext des in Kapitel 3.2.7 vorgestellten [Abstract Windowing Toolkit \(AWT\)](#) ergeben sich gute reale Anwendungsfälle für anonyme innere Klassen.

Abschlußbemerkung:

Die weiteren Spielarten innerer Klassendefinitionen werden wegen der geringen praktischen Relevanz -- außerhalb des AWT --, und tendenziellen Unübersichtlichkeit des entstehenden Entwurfs nicht diskutiert. Eine gute Referenz findet sich im für Ausbildungszwecke (als HTML) kostenfrei verfügbaren Buch [GoTo Java2](#).

Mit [strictfp](#) existiert seit Java v1.2 ein neues Schlüsselwort zur Modifikation des Klassenverhaltens. Es deaktiviert die erweiterte Gleitpunktdarstellung nach [IEEE 754-1985](#). In diesem Modus werden `float`-Datentypen statt mit 32 mit 43-Bit, bzw. `double`-Datentypen mit 79

statt 64-Bit behandelt.

Durch die bessere Nutzung der vorhandenen Zielhardware ergibt sich neben Performancegewinnen auch eine erhöhte Berechnungsgenauigkeit. Als Resultat kann derselbe Code auf verschiedenen realen Maschinen, je nach Auslegung der Gleitkommaarithmetik, zu verschiedenen Berechnungsergebnissen führen.

Durch Angabe von `strictfp` wird somit wieder portables Verhalten der Fließkommaoperationen erzwungen.

Das Beispiel (nach Kazuyuki SHUDO) zeigt die Verwendung dieses Schlüsselwortes:

```
(1)class StrictfpTest {
(2)  private static double defaultDmul(double a, double b) {
(3)    return a * b;
(4)  }
(5)
(6)  private static strictfp double strictDmul(double a, double b) {
(7)    return a * b;
(8)  }
(9)
(10) private static double defaultDdiv(double a, double b) {
(11)   return a / b;
(12) }
(13)
(14) private static strictfp double strictDdiv(double a, double b) {
(15)   return a / b;
(16) }
(17)
(18) public static void main(String[] args) {
(19)   double a, b, c;
(20)
(21)   /* multiplication */
(22)   a = Double.longBitsToDouble(0x0008008000000000L);
(23)   b = Double.longBitsToDouble(0x3ff0000000000001L);
(24)
(25)   System.out.println(a + " (0x0008008000000000)");
(26)   System.out.println(" * " + b + " (0x3ff0000000000001)");
(27)
(28)   c = defaultDmul(a, b);
(29)   System.out.println("default : " + c +
(30)     " (0x" + Long.toHexString(Double.doubleToLongBits(c)) + ")");
(31)
(32)   c = strictDmul(a, b);
(33)   System.out.println("strictfp: " + c +
(34)     " (0x" + Long.toHexString(Double.doubleToLongBits(c)) + ")");
(35)
(36)   System.out.println();
(37)
(38)   /* division */
(39)   a = Double.longBitsToDouble(0x000fffffffffffffffL);
(40)   b = Double.longBitsToDouble(0x3fefffffffffffffffL);
(41)
(42)   System.out.println(a + " (0x000fffffffffffffff)");
(43)   System.out.println(" / " + b + " (0x3fefffffffffffffff)");
(44)
(45)   c = defaultDdiv(a, b);
(46)   System.out.println("default : " + c +
(47)     " (0x" + Long.toHexString(Double.doubleToLongBits(c)) + ")");
(48)
(49)   c = strictDdiv(a, b);
(50)   System.out.println("strictfp: " + c +
(51)     " (0x" + Long.toHexString(Double.doubleToLongBits(c)) + ")");
(52) }
(53)}
```

Beispiel 31: Verwendung des Schlüsselwortes `strictfp` [StrictfpTest.java](#)

Die Unterstützung durch die virtuelle Maschine vorausgesetzt, liefert die Ausführung des Programms folgende Ausgabe:

1.112808544714844E-308 (0x0008008000000000) * 1.0000000000000002 (0x3ff0000000000001)


```

default : 1.112808544714844E-308 (0x80080000000000)
strictfp: 1.112808544714844E-308 (0x80080000000001)

2.225073858507201E-308 (0x000fffffffffffffff) / 0.9999999999999999 (0x3fefffffffffffffff)
default : 2.2250738585072014E-308 (0x100000000000000)
strictfp: 2.225073858507201E-308 (0xfffffffffffffff)

```

SUNs Referenzimplementierung der virtuellen Maschine unterstützen ab Version 1.3 auf den Intelplattformen Windows und Linux die korrekte Umsetzung des erweiterten Fließkommaformates.

Objekterzeugung:

Die Erzeugung konkreter Ausprägungen (auch: Instanzen oder Objekte) einer Klasse geschieht üblicherweise durch den new-Operator.

Die Syntax lautet: *Type Variable = new Type(Parameters)*

Wird ein Objekt nicht mehr durch den Programmierer referenziert, so wird es durch den Garbage Collector aus dem Speicher entfernt. Dies kann u. U. auch erst verzögert geschehen, da der Garbage Collector als eigener asynchroner Thread der virtuellen Maschine realisiert ist. (Der Aufruf [System.gc\(\)](#) schlägt der virtuellen Maschine vor den Garbage Collector bei Gelegenheit aufzurufen.)

▲ 2.4.3 Attribute

Attribute stellen in der objektorientierten Programmierung den üblichen -- und bei strenger Betrachtung den einzigen -- Weg zur Ablage dynamischer Zustandsinformation eines Objekts dar. (Selbstverständlich können auch Variablen innerhalb von [Methoden](#) beliebige Informationen aufnehmen. Jedoch sind diese Inhalte im Allgemeinen nach Verlassen des Gültigkeitsbereichs verloren).

[Abbildung 2](#) stellt die Platzierung der Attribute in UML-Notation dar.

Vereinfachte **Syntax** einer Attributdefinition:

```

(public,
protected,
private,
static,
final,
transient,
volatile)zero or more Identifizier []optional (= Initialization)optional

```

Die Syntaxkomponenten im Einzelnen:

Syntaxelement Semantik

public	Das Attribut ist allgemein für alle anderen Klassen sichtbar.
protected	Das Attribut ist ausschließlich für die definierende Klasse sowie diejenigen sichtbar die von aktuellen erben, sowie alle Klassen desselben Packages.
private	Das Attribut ist nur innerhalb der deklarierenden Klasse sichtbar.
static	Gültigkeitsbereich des Attributs ist die gesamte Klasse. D.h. alle Objekte dieser Klasse teilen sich <i>dasselbe</i> Attribut (= derselbe Speicherplatz). Ein so deklariertes Attribut hat in allen Ausprägungen denselben Wert; Änderungen finden automatisch synchronisiert in allen Objekten statt. Ohne Angabe dieses Schlüsselwortes ist der Scope auf die Objektebene festgelegt.
final	Das Attribut ist konstant, und kann nach seiner (zwingend anzugebenden!) Initialisierung nicht mehr verändert werden.
transient	So ausgezeichnete Attribute sind nicht Teil des persistenten Objektzustandes, und werden bei einer Ablage auf Sekundärspeicher (File, Datenbank, etc.) ignoriert.
volatile	Kennzeichnet ein Attribut als bei Optimierungen nicht zu berücksichtigen. Die Semantik und Anwendung ist ähnlich zur aus C/C++ bekannten Mimik.

```

(1)public class Attributes {
(2)    int          testAttribute1;
(3)    public short testAttribute2 = 99;
(4)    private     long          testAttribute3 = 5L;
(5)    protected boolean testAttribute4;
(6)    volatile    private char testAttribute5;
(7)    public transient long testAttribute6 = 0x42;
(8)
(9)    static final double pi = 3.141592654;
(10)
(11)    TestClass testAttributeC1;
(12)    TestClass testAttributeC2 = new TestClass();
(13)} //class Attributes
(14)
(15)class TestClass {
(16)} //class TestClass

```

Beispiel 32: Einige Attributdefinitionen [Attributes.java](#)

Abschlußbemerkungen:

- Aus Gründen der Speicherplatzgründen empfiehlt sich die Definition von Konstanten (wie pi im Beispiel) als `static final`.
- Java erlaubt, anders als C++, keinerlei Datenstrukturen außerhalb von Klassen. Globale Variablen, sowie wiederverwendbare „Allzweckvariablen“ -- beispielsweise als Schleifenzähler -- können daher nicht in der gewohnten (Un-)Sitte realisiert werden.

▲ 2.4.4 Operationen und Methoden

Operationen und Methoden bilden das dynamische Verhalten eines Objekts ab. Während der Begriff *Operation* nur die Signatur, bestehend aus Rückgabotyp, Operationsnamen und der Parameterliste, bezeichnet, deckt *Methode* auch die programmiersprachliche Umsetzung ab.

Java erlaubt ausschließlich die Definition von Methoden innerhalb von Klassen. Globale Funktionen sind ebenso wie globale Variablen nicht möglich!

Vereinfachte **Syntax** einer Operation:

```

(public,
protected,
private,
abstract,
static,           ResultType  Identifier  (ParameterList)  (throws ExceptionList)optional
final,
synchronized,
native,
strictfp)zero or more

```

Die Syntaxkomponenten im Einzelnen:

Syntaxelement Semantik

public	Die Methode ist allgemein für alle anderen Klassen sichtbar.
protected	Die Methode ist ausschließlich für Klassen sichtbar die von der aktuellen erben, sowie alle Klassen desselben Packages.
private	Die Methode ist nur innerhalb der deklarierenden Klasse sichtbar.
abstract	Definiert analog zum Schlüsselwort auf Klassenebene, daß eine so gekennzeichnete Operation keine Implementierung bereitstellt; mithin keine Methode implementiert.
static	Gültigkeitsbereich dieser Methode ist die Klasse. In der Anwendung bedeutet dies, daß eine so deklarierte Methode sowohl auf der Klasse selbst, als auch einem Objekt dieser Klasse aufgerufen werden kann. Beispiel

<code>final</code>	Die Methode darf nicht in einer ererbten Klasse überschrieben werden.
<code>synchronized</code>	Auf einer so deklarierten Methode wird durch das Laufzeitsystem ein wechselseitiger Ausschluß realisiert. Das bedeutet, daß sich nur jeweils ein Thread innerhalb einer Methode des Objektes befinden darf.
<code>native</code>	Eine so gekennzeichnete Methode ist nicht in Java realisiert, sondern wird als nativer Code einer anderen Programmiersprache (z.B. C/C++) realisiert. Das JDK bietet für die beiden genannten Sprachen Unterstützung in Form automatisch generierter Headerdateien an.
<code>strictfp</code>	Deaktiviert Nutzung plattformspezifischer Gleitkommahardware. analog strictfp auf Klassenebene

Aufgrund der strengen Typisierung der Programmiersprache ist jede Javaoperation über ihren Rückgabewert typisiert. Als **Rückgabetypen** stehen alle [primitiven built-in Datentypen](#), alle [Klassen](#) und [Schnittstellen](#), sowie der explizite *Nichttypvoid* zur Verfügung.

Wie aus C/C++ bekannt geben void-Methoden keine Werte an den Aufrufer zurück, und können daher nicht innerhalb von Ausdrücken verwendet werden.

Ebenso analog der bekannten Mimik wird die explizite Rückgabe eines Wertes durch das Schlüsselwort `return` eingeleitet. Wie in (neuen) C/C++-Übersetzern üblich, prüft auch Java die Existenz einer erreichbaren typkompatiblen Return-Anweisung innerhalb des Methodenrumpfes.

Die auf den Operationsnamen folgende **Parameterliste** setzt sich aus einer Folge von Typen und Parameternamen zusammen, kann jedoch auch leer sein.

Verursacht durch das Fehlen expliziter Referenztypen (wie Zeiger) werden alle Parameter, die [primitive built-in Typen](#) sind **by value** und alle objektartigen Parameter per Vorgabe **by reference** übergeben.

Hinweis: Es existiert kein Mechanismus dieses Verhalten zu überschreiben oder abzuändern!

Lediglich für die by reference Interpretation der Primitivtypen ist mit den [Wrapper Typen](#) eine Standardmethodik vorgesehen.

Genaugenommen kommt auch für Objekte, die als Parameter einer Methode verwendet werden, eine Wertübergabe zum Einsatz. Jedoch wird in diesem Falle nicht der Wert des Objektes übergeben, d.h. es wird keine Kopie des Objektes erzeugt und an die aufzurufende Methode übergeben. Vielmehr wird eine Kopie der Referenz auf das Objekt übergeben.

Dieser Umstand führt dazu, daß auch objektwertige Parameter innerhalb eines Methodenrumpfes nicht direkt modifiziert werden können, sondern hierfür auf Methoden des zu verändernden Objekts zurückgegriffen werden muß.

[Mehr Informationen hierzu.](#)

Die **Signatur einer Javamethode** wird aus Operationsname und der Parameterliste, unter Berücksichtigung der Parameterreihenfolge, gebildet.

Sichtbarkeitsattribute, ebenso wie der Rückgabetyt und weitere Eigenschaften gehen nicht in die Signaturbildung ein.

Innerhalb des **Methodenrumpfs** sind alle aus 2.3 bekannten [Kontrollstrukturen](#) zugelassen.

Darüberhinaus kann an beliebiger Stelle die Deklaration lokaler Variablen erfolgen. Im Gegensatz zu C/C++ ist die Lebensdauer lokaler Variablen strikt an die des umgebenden Blocks gebunden.

Daher steht das Schlüsselwort `static` für die Variablendefinition innerhalb von Methoden nicht zur Verfügung.

Der Aufruf von Methoden erfolgt in der bekannten *Punktnotation*, die in allgemeiner Form als `Objekt.Methode(Parameterliste)` oder entsprechend `Klasse.Methode(Parameterliste)` bei Klassenmethoden beschrieben werden kann.

Eine Sonderrolle spielt das **Schlüsselwort this im Methodenrumpf**. Es steht als impliziter Parameter in allen nicht-statischen Methoden zur Verfügung. `this` referenziert immer das aktuelle Objekt. Die lokale Variable `this` ist dabei immer von Typ [final](#) und erlaubt keine Neuzuweisungen. Üblicherweise ist die Verwendung von `this` nicht explizit erforderlich. Zur Behebung von Namenskonflikten zwischen Übergabeparametern und lokalen Variablen leistet es jedoch wertvolle Dienste. Weitere Anwendung findet das Schlüsselwort zur [Aufruf von Konstruktoren](#).

```

(1)public class ThisDemo {
(2)    private int i = 42;
(3)
(4)    public void setI(int i) {
(5)        this.i = i;
(6)    } //setI()
(7)
(8)    public int getI() {
(9)        return this.i;
(10)    } //getI()
(11)
(12)    public static void main(String[] args) {
(13)        ThisDemo td = new ThisDemo();
(14)        System.out.println("value of i="+td.getI() );
(15)        td.setI(45);
(16)        System.out.println("value of i="+td.getI() );
(17)    } //main()
(18)} //class ThisDemo

```

Beispiel 33: Zugriff auf Objekteigenschaften mit this [ThisDemo.java](#)

Bildschirmausgabe:

```

$java ThisDemo
value of i=42
value of i=45

```

Das Beispiel zeigt die Verwendung der Eigenobjektreferenz `this` innerhalb der Methoden `setI` und `getI`.

Während die Referenzierung von `i` innerhalb `getI` auch ohne vorstelltes explizites `this` eindeutig auflösbar ist, muß in `setI` das Schlüsselwort zwingend angegeben werden um dem Namenskonflikt zwischen dem Parameter `i` und dem gleichnamigen Attribut aufzulösen.

Analog zu den Klassenattributen werden durch das Schlüsselwort **static** **Klassenmethoden** definiert.

Auf sie kann unabhängig von der Existenz konkreter Objekte der Klasse zugegriffen werden.

Aufgrund ihrer instanzenunabhängigen Natur besitzen statische Methoden keine `this` Referenz, da ein so zu referenzierendes Objekt nicht existiert.

```

(1)public class StaticMethodTest {
(2)    public static void helloWorld() {
(3)        System.out.println("hello world");
(4)    } //helloWorld()
(5)
(6)    public static void main(String[] args) {
(7)        StaticMethodTest.helloWorld();
(8)        (new StaticMethodTest()).helloWorld();
(9)    } //main()
(10)} //class StaticMethodTest

```

Beispiel 34: Statische Methoden [StaticMethodTest.java](#)

Besondere Methoden:

Konstruktoren

Identisch benannt zur beherbergenden Klasse

Syntaktisch ähneln die Konstruktoren den „normalen“ Operationsdeklarationen. Jedoch mit dem Unterschied, daß kein Rückgabetypp spezifiziert werden kann.

Der Konstruktorenaufruf wird automatisch durch den Übersetzer plaziert.

Existiert kein expliziter Konstruktor, so wird während des Übersetzungsvorganges ein unparametrisierter Vorgabekonstruktor erzeugt. Dieser enthält keinerlei eigene Funktionalität, sondern ruft nur den entsprechenden parameterlosen Konstruktor der Superklasse auf. Verfügt eine Klasse hingegen ausschließlich über parametrisierte Konstruktoren, so wird kein unparametrisierter Defaultkonstruktor angelegt.

Aufrufe unter den verschiedenen Konstruktoren können durch das bekannte **Schlüsselwort this** vorgenommen werden. Hierbei werden die einzelnen Konstruktoren wie normale Methoden behandelt. Der Aufruf erfolgt durch `this` gefolgt von den Konstruktorenparametern in runden

Klammern. Der kaskadierende Konstruktorenaufruf muß zwingend als erstes Statement des Anweisungsblockes plaziert werden.
Zwar verfügt die Konstruktormethode über keinen expliziten Rückgabebetyp, wird aber intern als void-Typisiert behandelt.

```
(1)public class Construct {
(2)    public Construct() {
(3)        System.out.println("object created using explicitly stated
parameterless Constructor");
(4)    } //constructor
(5)
(6)    public Construct(int i) {
(7)        System.out.println("object Constructed by parametrized Constructor,
parameter i="+i);
(8)    } //constructor
(9)
(10)   public static void main(String args[]) {
(11)       new Construct();
(12)       new Construct(1);
(13)       new TestClass1();
(14)   } //main()
(15)} //class Construct
(16)
(17)class TestClass1 extends TestClass2 {
(18)    //nothing!
(19)} //TestClass1
(20)
(21)class TestClass2 {
(22)    TestClass2() {
(23)        this(1);
(24)        System.out.println("Constructor of TestClass2 called");
(25)    } //constructor
(26)
(27)    TestClass2(int i) {
(28)        this ((short) 2, (byte) 3);
(29)        System.out.println("Constructor of TestClass2 called paramter i="+i);
(30)    } //constructor
(31)
(32)    TestClass2(short s, byte b) {
(33)        System.out.println("Constructor of TestClass2 called paramter s="+s
+" parameter b="+b);
(34)    } //constructor
(35)} //class TestClass2
```

Beispiel 35: Verschiedene Konstruktoren [Construct.java](#)

Nicht-öffentliche Konstruktoren und Fabriken:

Häufig anzutreffen sind Klassen, die zwar einen explizit definierten Konstruktor bieten, diesen jedoch `private` deklarieren. In der Konsequenz ist eine Objekterzeugung per `new` nicht möglich. Zumeist wird dieser Ansatz angewandt, wenn die Objekterzeugung aufwendig ist und nicht dem Anwender überlassen werden kann oder soll.

Um dennoch Objekte der betreffenden Klasse erzeugen zu können wird eine statische *Fabrik*-Methode eingeführt, welche die Objekterzeugung übernimmt und implizit `new` aufruft.

```
(1)public class Construct2 {
(2)    private Construct2() {
(3)        System.out.println("hello world");
(4)    } //constructor
(5)
(6)    public static void main(String[] args) {
(7)        new Construct2();
(8)        /* new OtherClass(); would fail since constructor is declared to be
private */
(9)        OtherClass oc = OtherClass.OtherClassFactory(); //object creation
using simple factory approach
(10)    } //main()
(11)} //class Construct2
(12)
(13)class OtherClass {
(14)    public static OtherClass OtherClassFactory() {
(15)        return new OtherClass();
```

```

(16)    } //constructor
(17)
(18)    private OtherClass() {
(19)        System.out.println("other class created");
(20)    } //constructor
(21)} //class OtherClass

```

Beispiel 36: Nicht-öffentliche Konstruktoren und Fabrikmethode zur Objekterzeugung [Construct2.java](#)

Bildschirmausgabe:

```

$java Construct2
hello world
other class created

```

Der Konstruktorencall innerhalb der Klasse construct2 kann bei Absetzen des new ausgeführt werden, da aus der Klasse heraus (konkret: innerhalb einer der statischen Methode main) ein Objekt dieser Klasse erzeugt wird -- der private Konstruktor ist somit zugreifbar.

Hingegen ist die Erzeugung eines Objekts von otherClass per new nicht möglich, da der dort definierte private Konstruktor nicht aus construct2 heraus referenzierbar ist. Der entsprechende Fehler wird bereits zum Übersetzungszeitpunkt erkannt.

Die Erzeugung von Objekten der Klasse otherClass ist ausschließlich über die statische Methode otherClassFactory möglich. Sie ruft intern den privaten Konstruktor auf, der an dieser Stelle sichtbar und zugreifbar ist.

(siehe [Java API Specification](#), [siehe Java Language Specification](#))

Ein Beispiel für eine Klasse, die weder über eine Factorymethode, noch über öffentliche Konstruktoren verfügt ist die Standard-API-Klasse [Void](#).

Ihre Implementierung ist in der API wie folgt festgelegt:

```

public final
class Void {
    public static final Class TYPE = Class.getPrimitiveClass("void");

    private Void() {}
}

```

Noch vor dem Konstruktor werden die [statischen Initialisierungen](#) aufgerufen. Aufgrund der Reihenfolge in der Methodenabarbeitung existiert zum Ablaufzeitpunkt der statischen Initialisierungen das zu erzeugende Objekt noch nicht; der static-Block wird quasi zum Ladezeitpunkt der Klasse ausgeführt. Daher können zu diesem Zeitpunkt keine Zugriffe auf nichtstatische Speicherobjekte (Methoden und Attribute) erfolgen. Zugriffe auf statische Attribute und Methoden sind jedoch möglich.

Der Initialisierungsblock darf nicht durch Unterbrechungsanweisungen wie break oder return vorzeitig verlassen werden. Hierunter fallen auch Exceptions, die zum vorzeitigen Verlassen des Anweisungsblockes führen würden.

```

(1)public class StaticInit {
(2)    static int i=42;
(3)
(4)    static {
(5)        System.out.println("value of i="+i);
(6)        helloWorld();
(7)        i = 43;
(8)    } //static
(9)
(10)   public static void helloWorld() {
(11)       System.out.println("hello world");
(12)   } //helloWorld()
(13)
(14)   public static void main(String[] args) {
(15)       StaticInit myObj = new StaticInit();
(16)   } //main()
(17)
(18)   public StaticInit() {
(19)       System.out.println("value of i="+i);
(20)   } //constructor
(21)} //class StaticInit

```

Beispiel 37: Statische Initialisierung [StaticInit.java](#)**BildschirmAusgabe:**

```
$java StaticInit
value of i=42
hello world
value of i=43
```

Das Beispiel zeigt zunächst den Zugriff auf das statische Attribut (Klassenattribut) `i` direkt nach dessen Definition und Initialisierung. Zu diesem Zeitpunkt ist der Initialisierungswert 42 gesetzt. Der Aufruf von `helloWorld` demonstriert die Möglichkeit vor der Objekterzeugung statische Methoden auszuführen.

Nach Abarbeitung der statischen Initialisierung wird der Konstruktor bearbeitet. In ihm steht der Wert von `i` nur noch in der durch die statische Initialisierung modifizierten Form zur Verfügung.

Ausführungsreihenfolge der konstruierenden Codesequenzen:

(als Pseudocode):

```
for-each (Superclass s) {
    s.AttributeInitialization()
    s.StaticBlock()
} //for-each
for-each (Superclass s) {
    s.Constructor()
}
```

Beginnend mit der hierarchiehöchsten Superklasse werden zunächst die Initialisierungen der Attribute, im Anschluß daran (falls vorhanden) der static-Block dieser Klasse, abgearbeitet. In einem zweiten Durchlauf über die Klassenhierarchie werden die Konstruktoren der Superklassen in derselben Reihenfolge zur Ausführung gebracht.

```
(1)public class SuperClassConstruct extends Class2 {
(2)    static int i=6;
(3)
(4)    static {
(5)        System.out.println("i="+i--);
(6)        System.out.println("static initialization of class
SuperClassConstruct");
(7)        System.out.println("i="+i);
(8)    } //static
(9)
(10)   public SuperClassConstruct() {
(11)       System.out.println("object of class SuperClassConstruct
constructed");
(12)   } //constructor
(13)
(14)   public static void main(String[] args) {
(15)       new SuperClassConstruct();
(16)   } //main()
(17)} //class SuperClassConstruct
(18)
(19)class Class2 extends Class1 {
(20)    static int i=4;
(21)
(22)    static {
(23)        System.out.println("i="+i--);
(24)        System.out.println("static initialization of class Class2");
(25)        System.out.println("i="+i);
(26)    } //static
(27)
(28)    public Class2() {
(29)        System.out.println("object of class Class2 constructed");
(30)    } //constructor
(31)} //class Class2
(32)
(33)class Class1 {
(34)    static int i=2;
(35)    static {
```

```

(36)          System.out.println("i="+i--);
(37)          System.out.println("static initialization of class Class1");
(38)          System.out.println("i="+i);
(39)      } //static
(40)
(41)      public Class1() {
(42)          System.out.println("object of class Class1 constructed");
(43)      } //constructor
(44)} //class Class2

```

Beispiel 38: Initialisierungsreihenfolge [SuperClassConstruct.java](#)

BildschirmAusgabe:

```

$java SuperClassConstruct
i=2
static initialization of class class1
i=1
i=4
static initialization of class class2
i=3
i=6
static initialization of class superClassConstruct
i=5
object of class class1 constructed
object of class class2 constructed
object of class superClassConstruct constructed

```

Destruktoren --finalize

Zwar können in Java, anders als in C++, Objekte nicht durch Destruktorenaufruf zerstört werden, sondern nur durch Null-Zuweisung als nicht mehr benötigt markiert werden, Destruktoren werden jedoch weiterhin explizit zur Verfügung gestellt.

Destruktoren werden nicht durch den Anwender aufgerufen, sondern implizit erst zum Zerstörungszeitpunkt eines Objekts. Genaugenommen ist die Ausführung des Destruktors nicht garantiert. Terminiert die Applikation vor dem Ablauf des [Garbage Collectors](#), so werden die Objekte implizit durch das Betriebssystem aus dem Speicher entfernt, ohne das die Java-Laufzeitumgebung zunächst die finalize-Methoden aufruft.

Die Sichtbarkeitseinschränkung von finalize muß mindestens protected sein. Dies rührt von der impliziten Überschreibung der von Object ererbten Methode finalize her. (vgl. [java.lang.Object:finalize](#))

```

(1)public class DestruktorTest {
(2)    public static void main(String[] args) {
(3)        Test t1 = new Test();
(4)        t1 = null;
(5)        System.gc();
(6)    } //main()
(7)} //class DestruktorTest
(8)
(9)class Test {
(10)    public void finalize() {
(11)        System.out.println("destructor of class Test called");
(12)    } //finalize()
(13)} //class Test

```

Beispiel 39: Destruktorenaufruf durch Garbage Collector [DestruktorTest.java](#)

Im Beispiel wird das Objekt t1 zunächst durch Nullsetzung zum Löschen markiert, und im anschließenden Garbage Collector-Aufruf gelöscht. Während des Entfernens aus dem Speicher wird die dargestellte Nachricht am Bildschirm ausgegeben.

Hauptmethode -- main

Pro Java-Applikation kann maximal eine Methode der Signatur main(String[]) existieren. Sie wird beim Startvorgang innerhalb der öffentlichen Klasse gesucht, die namensgebend für die Klassendatei ist.

Applets verfügen hingegen über keine automatisch aufgerufene main-Methode, sondern werden durch init() Initialisiert und durch das im Anschluß darauf aufgerufene start() ausgeführt.

Die Methode `toString`

ist auf der Superklasse [Object](#) aller Javaklassen definiert. Sie wird von allen Klassen der Standard-API implementiert. Durch sie wird immer eine Zeichenkettenrepräsentation des aktuellen Objekts zurückgegeben.

Diese Methode wird standardmäßig bei der Ausgabe per `System.out.println` aufgerufen.

Zum Abschluß: Sichtbarkeit von Attributen und Methoden im Überblick:

	definierende Klasse	Abgeleitete Klasse	Package	Alle anderen
<i>default</i> (keine explizite Festlegung)	✓			
public	✓	✓	✓	✓
private	✓			
protected	✓	✓ Zugriff auf Attribute und Methoden der Subklasse	✓	

Anmerkung: Der Zugriff auf als `protected` deklarierte Attribute und Methoden ist auch über Objektreferenzen möglich, die denselben Typ haben wie die definierende Klasse.

▲ 2.4.5 Aufzählungstypen

Ab Version 1.5 führt Java mit dem Schlüsselwort `enum` einen eigenständigen und expliziten Mechanismus zur Definition von Aufzählungstypen ein.

Konzeptionell sind Aufzählungstypen identisch zu Variablen, die innerhalb des Rumpfes einer Methode deklariert werden oder Variablen und Attributen einer Klasse gleichgestellt. Aus diesem Grunde ähnelt die Definitionssyntax auch den bekannten Darstellungsformen:

Vereinfachte **Syntax** der Enum-Definition:

```
(public,
protected,
private)_one of      staticopt      finalopt      enum      Identifier      {value}
```

Im einfachsten Anwendungsfall besteht die Definition eines Aufzählungstypen aus der durch Kommata voneinander separierten vollständigen Aufzählung aller Werte.

Das Beispiel zeigt die Bildung des Aufzählungstyps `season`, der durch die zulässigen Werte `winter`, `spring`, `summer` und `fall` konstituiert wird.

```
(1)public class EnumTest1 {
(2)    public static void main(String[] args) {
(3)        enum season { winter, spring, summer, fall; }
(4)
(5)        season s1 = season.winter;
(6)
(7)        if (s1 == season.winter)
(8)            System.out.println("It's "+s1);
(9)
(10)       season s2 = season.winter;
(11)    } //main()
(12)} //class EnumTest1
```

Beispiel 40: Definition eines einfachen Aufzählungstyps [EnumTest1.java](#)

Die Verwendung von Aufzählungstypen entspricht der von primitiven Typen. Aus diesem Grund ist keine Anforderung von Speicherplatz durch das `new`-Schlüsselwort notwendig. Der Übersetzer verhindert sogar aktiv die Instanziierung eines Aufzählungstyps via `new` und bricht beim Versuch mit der Fehlermeldung `enum types may not be instantiated` ab.

Im Gegensatz zu Ausprägungen der Primitivtypen besitzen Aufzählungsinstanzen jedoch keinen Vorgabewert mit dem sie standardmäßig initialisiert werden. Stattdessen führt die Verwendung einer nichtinitialisierten Ausprägung eines Aufzählungstypen zu einem Übersetzungsfehler (Fehlermeldung: `variable ... might not have been initialized`).

Ansonsten bietet die Umsetzung die für Primitivtypen bekannten Eigenschaften. So liefert die Ausgabe diejenige Zeichenkette (d.h. den Wert) mit dem Aufzählungsinstanz belegt wurde. Ebenfalls identisch zu den Primitivtypen besitzen Aufzählungsinstanzen keine Identität. Dies äußert sich darin, daß zwei mit demselben Wert belegte Aufzählungen als identisch betrachtet werden.

Nicht identisch sind hingegen Ausprägungen verschiedener Aufzählungstypen, die vermeintlich denselben Wert enthalten, d.h. in deren zur Definition verwendeten Werteliste sich lexikalisch dieselben Einträge finden.

So würde die nachfolgende Zuweisung bereits durch den Übersetzer (mit der Fehlermeldung `incompatible types`) abgelehnt

```
enum seasonE { winter, spring, summer, fall; };
enum seasonG { winter, frühling, sommer, herbst; };
seasonE s = seasonG.winter;
```

Dasselbe gilt auch für den Versuch des Vergleichs der Inhalte zweiter Aufzählungsausprägungen, wie sie durch das nachstehende Codefragment versucht wird.

```
enum seasonE { winter, spring, summer, fall; };
enum seasonG { winter, frühling, sommer, herbst; };
seasonE s1 = seasonE.winter;
seasonG s2 = seasonG.winter;
if (s1==s2) ...
```

Auch in diesem Fall wird bereits zum Übersetzungszeitpunkt durch die Fehlermeldung `incomparable types: seasonG and seasonE` auf den Fehler hingewiesen.

In Erweiterung der bisher vorgestellten Syntax lassen sich Aufzählungstypen sogar „klassenartig“ ausbauen. Diese Erweiterung erlaubt es die Elemente des Aufzählungstypen wahlfrei an selbstdefinierte Eigenschaften zu binden. Konzeptionell werden diese Eigenschaften dabei als Attribute des Aufzählungstypen aufgefaßt, die durch einen durch den Programmierer bereitzustellenden Konstruktor zugewiesen werden. Der Konstruktoreaufruf erfolgt dabei automatisch durch das Laufzeitsystem zum Definitionszeitpunkt eines Aufzählungstypen für alle konstituierenden Inhaltselemente. Das nachfolgende Beispiel zeigt eine Verwendung des erweiterten Konzepts:

```
(1)public class EnumTest2 {
(2)    public enum Coin {
(3)        penny(1), nickel(5), dime(10), quarter(25);
(4)        private int value;
(5)        Coin(int value) {
(6)            System.out.println("creating: "+value);
(7)            this.value = value;
(8)        } //constructor
(9)        public int value() {
(10)            return value;
(11)        } //value()
(12)    } //enum Coin
(13)
(14)    public static void main(String[] args) {
(15)        Coin c1 = Coin.dime;
(16)        System.out.println( c1.value() );
(17)    } //main
(18)} //class EnumTest2
```

Beispiel 41: Definition eines klassenartigen Aufzählungstyps [EnumTest2.java](#)

Das Beispiel definiert den Typ `Coin` mit seinen Inhaltstypen `penny`, `nickel`, `dime` und `quarter`. Den Inhaltstypen wird durch Konstruktoreaufruf jeweils ein Inhaltswert zugeordnet. Dieser Wert wird der definierten privaten Variable `value` zugewiesen.

Durch die Methode `value` kann der dem jeweiligen Inhaltstyp zugewiesene Wert ausgelesen werden.

Methoden, die den Inhalt der definierten Variable schreiben können zwar definiert werden, jedoch werden Wertänderungen nicht auf die vordefinierten Inhaltstypen synchronisiert.

Die Anzahl der intern mit einem Wert verbundenen Festwerte ist hier bei nicht beschränkt. Das abschließende Beispiel zeigt die Zuweisung von zwei Einzelwerten im Konstruktor:

```
(1)public class EnumTest3 {
(2)    public enum Time {
(3)        highNoon(12,00), sunSet(20,30), sunRise(5,30), teaTime(17,00);
(4)    private int hour;
(5)        private int minute;
(6)
(7)        Time(int hour, int minute) {
(8)            this.hour = hour;
(9)            this.minute = minute;
(10)        } //constructor
(11)        public void printTime() {
(12)            System.out.print(hour+":"+minute);
(13)        } //printTime()
(14)    } //enum Time
(15)
(16)    public static void main(String[] args) {
(17)        Time t1 = Time.sunSet;
(18)        System.out.print( "Sunset is at: ");
(19)        t1.printTime();
(20)    } //end main
(21)} //class EnumTest3
```

Beispiel 42: Definition eines klassenartigen Aufzählungstyps [EnumTest3.java](#)

▲ 2.4.6 Wrapper-Typen

Korrespondierend zu jedem [primitiven Datentypen](#) in Java gibt es einen *Wrapper Typen*. Sie kapselt den zugrundelegenden Primitivtyp in einer eigenen Klasse, und stellt einige Servicemethoden bereit.

Objekte aller Wrappertypklassen können nur bei ihrer Erzeugung mit Werten versehen werden, die über die gesamte Lebensdauer nicht mehr verändert werden können.

Primitivtyp

[boolean](#)

[byte](#)

[short](#)

[int](#)

[long](#)

[float](#)

[double](#)

[void](#)

Dieser Typ ist nicht als Datentyp für Variablen und Attribute verfügbar, sondern kann nur als Rückgabebetyp von Operationen spezifiziert werden.

Wrapper Typ

[Boolean](#)

[Byte](#)

[Short](#)

[Integer](#)

[Long](#)

[Float](#)

[Double](#)

[Void](#)

Kann nicht instanziiert werden.

Die Abbildung 9 zeigt die Organisation der Wrapperklassen innerhalb der Standard-API im Überblick:

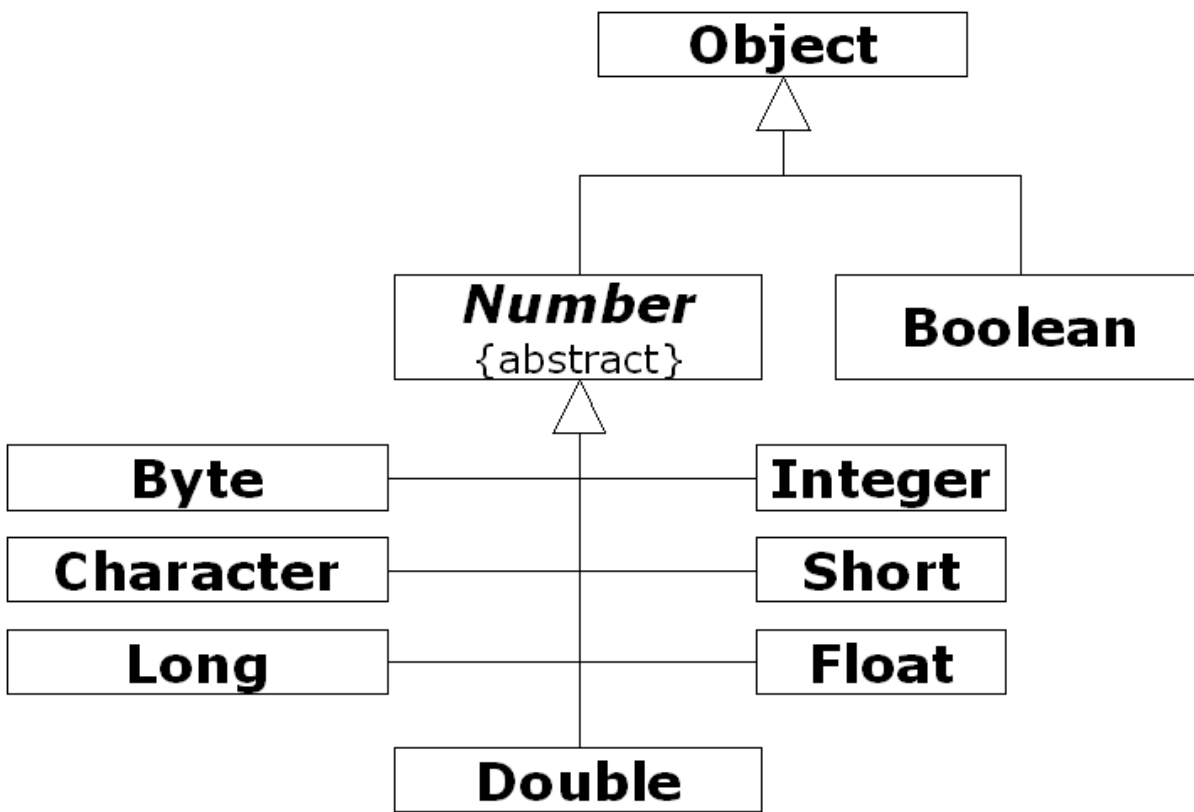


Abbildung 9: Hierarchie der Wrapperklassen (click on image to enlarge!)

```

(1)public class CBRCBV {
(2)    public static void main(String[] args) {
(3)        TestClass testObj = new TestClass();
(4)        int testVar;
(5)        Byte testWrapperType;
(6)
(7)        testObj.setS((short) 42);
(8)        testWrapperType = new Byte((byte) 12);
(9)        testVar = 50;
(10)
(11)        System.out.println("values before method run");
(12)        System.out.println("testVar = "+testVar);
(13)        System.out.println("testWrapperType = "+testWrapperType.byteValue
(14)    );
(15)        System.out.println("testObj.s = "+testObj.getS() );
(16)        aMethod(testVar, testWrapperType, testObj);
(17)
(18)        System.out.println("values after method run");
(19)        System.out.println("testVar = "+testVar);
(20)        System.out.println("testWrapperType = "+testWrapperType.byteValue
(21)    );
(22)        System.out.println("testObj.s = "+testObj.getS() );
(23)    } //main()
(24)    private static void aMethod(int i, Byte b, TestClass tc)    {
(25)        i++;
(26)        b = new Byte((byte) 0);
(27)        tc.setS( (short) (tc.getS()+1) );
(28)
(29)        System.out.println("values within method after modification:");
(30)        System.out.println("testVar = "+i);
(31)        System.out.println("testWrapperType = "+b.byteValue() );
(32)        System.out.println("testObj.s = "+tc.getS() );
(33)    } //aMethod()
(34)} //class CBRCBV
(35)
(36)class TestClass {
(37)    private short s;
(38)
(39)    public short getS() {
(40)        return s;

```

```

(41)    } //getS
(42)
(43)    public void setS(short s) {
(44)        this.s = s;
(45)    } //getS()
(46)} //TestClass

```

Beispiel 43: Verwendung von primitiven, Wrapper und objektwertigen Typen [CBRCBV.java](#)

BildschirmAusgabe:

```

values before method run
testVar = 50
testWrapperType = 12
testObj.s = 42
values within method after modification:
testVar = 51
testWrapperType = 0
testObj.s = 43
values after method run
testVar = 50
testWrapperType = 12
testObj.s = 43

```

Im Beispiel werden zunächst Exemplare der drei verschiedenen Typfamilien -- Primitivtyp, Wrapper Typ und Objekt -- erzeugt und mit Werten versehen.

Innerhalb der Methode `aMethod`, die alle drei Variablen als Übergabeparameter erhält, werden die Inhalte lokal geändert. Während die `int`-Variable direkt beschrieben werden kann, wird die Änderung des objektwertigen Parameters `tc` durch eine Methode der Klasse `TestClass` realisiert. Auf dem Wrapper Typen ist durch die Java-API keine Modifikationsroutine vorgesehen. Daher wird der übergebenen Referenz ein neues Objekt zugewiesen.

Nach Ausführung der Methode `aMethod` werden die Werte nochmals ausgegeben. Hier zeigt sich, daß auch für Objekte **kein echtes Call-by-Reference** zur Verfügung steht, sondern lediglich eine Kopie auf die Referenz übergeben wurde. Aus diesem Grunde sind die Änderungen sowohl am übergebenen Primitivtypen als auch an der Variable des Typs `Byte` verloren.

Als einzige Möglichkeit zur Realisierung von Modifikationen an einem Objekt erweisen sich die Methoden dieses Objekts.

Mehr Information hierzu:

- [Understanding that parameters are passed by value and not by reference](#)
- [Pass-by-value semantics in Java applications](#)

Alle Wrappertypen bieten bestimmte Servicemethoden an, die in der Praxis häufig auftretende Anforderungen erfüllen. Darunter fallen neben der Ausgabe des Wrappertypeninhaltes (d.h. des gekapselten Primitivwertes) auch Umsetzungen der auf Objekten nicht zur Verfügung stehenden Typumwandlungen (casts), sowie Methoden zur Erzeugung der Primitivrepräsentationen aus anderen als den Wrappertypendarstellungen (z.B. aus beliebigen Zeichenketten).

Wichtige und gebräuchliche Servicemethoden, sowie weitere Charakteristika der Wrappertypen:

- Alle Wrapperklassen sind `final` deklariert, und können daher nicht weiter spezialisiert werden.
- **...Value**
Beispiele: [booleanValue](#), [byteValue](#), [shortValue](#), [intValue](#), [longValue](#), [floatValue](#), [doubleValue](#)
Die abstrakte Signatur lautet in allen Fällen: `public T TValue()`, wobei `T` der jeweilige Primitivtyp ist (entsprechend für den Wrappertypen um `int`: `public int intValue()`).
Gibt den intern gekapselten Wert unverändert zurück.
Darüberhinaus existieren Methoden zur Ausgabe eines numerischen Wrappertypen in einen beliebigen [numerischen Primitivtypen](#). Diese ersetzen somit die explizite Typumwandlung.
- **Konstruktoren**
Erlauben die Erzeugung des Wrappertypen aus einem Ausdruck vom Typ des gekapselten Typen oder aus einem String. (Beispiele: [Boolean \(boolean value\)](#), [Boolean \(String s\)](#), [Integer \(int value\)](#), [Integer \(String s\)](#))
Ist dies nicht möglich so wird eine [NumberFormatException](#) ausgelöst. Dies ist der liegt vor, wenn der aus der Zeichenkette extrahierte Wert den zulässigen Gültigkeitsbereich (d.h. er liegt unter `MIN_VALUE` oder oberhalb `MAX_VALUE`) überschreitet. Ebenso wenn die Zeichenkette Symbole enthält, die nicht in eine numerische Darstellung überführt werden können.
- **compareTo**
Vergleicht den gekapselten Wert zweier Wrappertyp-Objekte.

Im Gegensatz hierzu vergleicht equals die Objekte direkt.

```
(1)public class WrapperComparison {
(2)    public static void main(String[] args) {
(3)        Integer i1 = new Integer ( 42 );
(4)        Integer i2 = new Integer ( 42 );
(5)
(6)        System.out.println("i1==i2="+ (i1==i2) );
(7)        System.out.println("i1.equals(i2)="+ (i1.equals(i2) ));
(8)        System.out.println("i1.compareTo(i2)="+ (i1.compareTo(i2)));
(9)    } //main()
(10)} //class WrapperComparison
```

Beispiel 44: Vergleich von Wrapperobjekten [WrapperComparison.java](#)

BildschirmAusgabe:

```
i1==i2=false
i1.equals(i2)=true
i1.compareTo(i2)=0
```

Hinweis:

Obwohl die Semantik der Operation equals für Objekte der Klasse Object und deren Subklassen als *größtmögliche unterscheidende Äquivalenzrelation* ([im Original](#)) festgelegt ist weicht z.B. die Implementierung in der Standard-API-Klasse String von dieser Maßgabe ab. Sie liefert bereits true wenn die beiden Zeichenketten inhaltsgleich sind, jedoch verschiedene Objekte darstellen.

- **parse...-Methode**

Die Fähigkeit der Konstruktoren, Wrapperobjekte -- nach vorheriger Gültigkeitsprüfung -- aus Zeichenketten zu erzeugen steht auch losgelöst von der Objektkonstruktion zur Verfügung. Auf allen numerischen Wrappertypen werden parseT-Methoden angeboten, wobei T den konkreten Primitivtypen bezeichne. Diese liefern, im Falle der Gültigkeit, aus einer Zeichenkette den jeweiligen Primitivtypen zurück.

Beispiele: [parseByte \(String s\)](#), [parseByte\(String s, int radix\)](#), [parseInt \(String s\)](#), [parseInt \(String s, int radix\)](#), etc.)

- **valueOf**

Für alle Wrappertypen [integraler Primitivtypen](#) ist mit der valueOf-Methode eine Factorymethode zur Erzeugung eines neuen Wrapperobjektes aus einer Zeichenkette definiert.

- Veränderbare Wrapperklassen für die primitiven Typen stehen mit den *CORBA Holdern* zur Verfügung.

(siehe [BooleanHolder](#), [byteHolder](#), [ShortHolder](#), [IntHolder](#), [LongHolder](#), [FloatHolder](#), [DoubleHolder](#)).

- Die üblichen arithmetischen Operationen stehen zur Verknüpfung von Wrapperobjekten nicht zur Verfügung (kein Operator overloading!).

▲ 2.4.7 Boxing/Unboxing

Zwar realisiert Java eine grundlegende Trennung zwischen Primitivtypen und Klassen, die Wertausprägungen nicht als Objekte auffaßt. Dennoch wird diese Maßgabe durch das in der Sprachversion 1.5 eingeführte dynamische Umwandlung zwischen primitiven Werten und den zugehörigen Wrapperklassen aufgeweicht. Die Integration dieses als *Boxing/Unboxing* bezeichnete Verfahren trägt den gesammelten Anwendererfahrungen Rechnung, die eine Vereinfachung der aufwendigen Wrapper-Objekterzeugung bzw. Wertextraktion aus bestehenden Wrapperobjekten fordern. Das namensgebende Begriffspaar bezeichnet hierbei diese beiden Schritte, d.h. *Boxing* die automatische Wrapper-Objekterzeugung bzw. *Unboxing* die Wandlung eines objektgekapselften Wertes in einen Primitivwert.

Die automatische Typwandlung wird in Java ausschließlich für Übergabeparameter angeboten, wie das nachfolgende Beispiel zeigt:

```

(1)public class BUBTest1 {
(2)    public static void main(String[] args) {
(3)        BUBTest1 obj = new BUBTest1();
(4)        obj.boxIt( new Integer(42) ); //nothing new
(5)        obj.boxIt( 42 ); //dynamic boxing
(6)
(7)        obj.unBoxIt( 42 ); //nothing new
(8)        obj.unBoxIt( new Integer(42) ); //dynamic unboxing
(9)    } //main()
(10)   public void boxIt(Integer i) {
(11)       System.out.println("value="+i);
(12)   } //boxIt
(13)   public void unBoxIt(int i) {
(14)       System.out.println("value="+i);
(15)   } //unBoxIt
(16)} //class BUBTest1

```

Beispiel 45: Dynamisches Boxing/Unboxing [BUBTest1.java](#)

Das Beispiel zeigt neben den jeweils singnaturkonformen Aufrufen auch die Verwendung der dynamischen Typwandlung; für die Methode `boxIt` die automatische Erzeugung eines innerhalb des Methodenrumpfes referenzierten Wrapper-Objekts bzw. für `unBoxIt` die Extraktion des im übergebenen Wrapper-Objekt enthaltenen `int` Primitivwertes.

Der Boxingvorgang kann außer für die Wrapper-Typen auch für Objekte der Klasse `Object` durchgeführt werden, da diese als Superklasse aller Wrapper-Typen angelegt sind ist diese Konversion stets typkonform. Das nachfolgende Beispiel zeigt die Anwendung:

```

(1)public class BUBTest2 {
(2)    public static void main(String[] args) {
(3)        BUBTest2 obj = new BUBTest2();
(4)        obj.boxIt( new Integer(42) ); //nothing new
(5)        obj.boxIt( 42 ); //dynamic boxing
(6)
(7)    } //main()
(8)    public void boxIt(Object i) {
(9)        System.out.println("value="+i);
(10)    } //boxIt
(11)} //class BUBTest2

```

Beispiel 46: Dynamisches Boxing für den Typ `Object` [BUBTest2.java](#)

Das Beispiel zeigt die bisher schon mögliche Verwendung einer Ausprägung des Typs `Integer` an einer Stelle, an der eine Ausprägung von `Object` erwartet wird, was unter Nutzung der Subklassenpolymorphie möglich ist. Ausgehend von diesem Sachstand ist die Realisierung des dynamischen Boxings im Beispiel zu verstehen. Das Auftreten der `int`-Zahl wird automatisch in eine Ausprägung der Klasse `Integer` konvertiert, die dann typkonform anstelle der erwarteten `Object`-Instanz verwendet werden kann.

Die umgekehrte Nutzung, d.h. die Verwendung einer Ausprägung von `Object` an einer Stelle, an der eine konkrete `int`-Zahl erwartet wird, ist --- nach Maßgabe der nicht typsicher möglichen Konversion entgegen der Vererbungsrichtung --- nicht möglich.

▲ 2.4.8 Vererbung

Wie in objektorientierten Sprachen üblich, unterstützt auch Java das Konzept der Vererbung. Die wesentlichen Hintergründe und Anwendungsgebiete, sowie Designaspekte sind aus früheren Lehrveranstaltungen bekannt, und werden daher an dieser Stelle nicht mehr wiederholt.

Charakteristika:

- Anders als C++ unterstützt Java **nur Einfachvererbung**!
- Syntaktisch wird Vererbung durch das Schlüsselwort `extends` ausgedrückt.
- Zur **Schnittstellenvererbung** (in C++ üblicherweise als *pure virtual class* realisiert) existiert mit den [Interfaces](#) ein explizites Sprachkonzept

- Subklasse erbt nicht private Methoden und Attribute der Superklasse.
- Vererbung wird implizit als **Spezialisierung** aufgefaßt.
Jede Ausprägung einer spezialisierten Klasse kann daher an jeder Stelle im Programm verwendet werden, an der eine Ausprägung der Superklasse erwartet wird (Liskov'sches Substitutionsprinzip). Hierbei erfolgt eine implizite Typumwandlung (*up cast*). Die originale Typinformation bleibt dessen ungeachtet erhalten.
- Umwandlung einer Subklassenausprägung in eine der Superklasse bedarf eines expliziten Cast-Statements.
Schlägt dieses fehl, wird eine [ClassCastException](#) ausgelöst (C++ retourniert hier ein null Objekt).

```
(1)public class InheritanceDemo {
(2)    public static void main(String[] args) {
(3)        C2 myC2 = new C2();
(4)
(5)        System.out.println("attrib2="+myC2.attrib2);
(6)        System.out.println("inherited attrib1="+myC2.attrib1);
(7)
(8)        C1 myC1 = myC2; //implicit type cast (up cast!)
(9)
(10)       C1 myC11 = new C1();
(11)
(12)       C2 myC22 = (C2) myC11; //explicit type cast needed (down cast!)
(13)                                           //
will throw a ClassCastException
(14)
(15)    } //main()
(16)} //class InheritanceDemo
(17)
(18)class C1 {
(19)    public int attrib1=1;
(20)} //class C1
(21)
(22)class C2 extends C1 {
(23)    public int attrib2=2;
(24)} //class C2
```

Beispiel 47: Erzeugung einer ClassCastException [InheritanceDemo.java](#)

Die Umwandlung des C2 Objektes in eines vom Typ C1 erfolgt implizit und automatisch.
Beim Versuch ein Objekt der Klasse c1 in ein Objekt der Klasse C2 umzuwandeln (*down cast* -- Subklassenobjekt wird in Superklassenobjekt gewandelt) wird eine ClassCastException erzeugt.

- Dynamische Bindung von nicht statischen Methoden (analog virtueller Funktionen in C++).
Dieses Verhalten kann durch Kennzeichnung der Methode als [final](#) unterbunden werden.
Klassenmethoden sind ebenso von diesem Verhalten ausgenommen.

```
(1)public class DynamicBinding {
(2)    public static void main(String[] args) {
(3)        C1 myC1 = new C1();
(4)        myC1.hello();
(5)
(6)        myC1 = new C2();
(7)        myC1.hello();
(8)
(9)        System.out.println("invoking sHello...");
(10)       ((C2) myC1).sHello();
(11)    } //main()
(12)} //class DynamicBinding
(13)
(14)class C1 {
(15)    public void hello() {
(16)        System.out.println("Hello from class C1");
(17)    } //hello()
(18)} //class C1
(19)
(20)class C2 extends C1 {
(21)    public void hello() {
(22)        System.out.println("Hello from class C2");
(23)    } //hello()
(24)
(25)    public void sHello() {
```



```

(26)         super.hello();
(27)         hello();
(28)     } //superHello()
(29)} //class C2

```

Beispiel 48: Dynamische Bindung von Methoden [DynamicBinding.java](#)

BildschirmAusgabe:

```

$java DynamicBinding
Hello from class C1
Hello from class C2
invoking sHello...
Hello from class C1
Hello from class C2

```

(Relevanter Teil der Ausgabe: oberhalb von *invoking sHello*) Trotz der Typisierung der Variable `myC1` als `C1`-Ausprägung wird die Methode `hello` von `C2` ausgeführt, wenn der Inhalt der Variable auf ein solches Objekt verweist.

- Das Schlüsselwort `super` dient als Platzhalter zur dynamischen Referenzierung des Superklassenobjekts.
Das vorhergehende Beispiel zeigt die Verwendung in der Methode `sHello`, die zunächst auf dem Superklassenobjekt die Methode `hello` ausführt, und im Anschluß die gleichnamige auf dem aktuellen Objekt.
Anmerkung: ein Zugriff auf die *Großelternklasse*, d.h. über mehrere Hierarchieebene hinweg -- etwa durch Konkatenation von `super` zu `super.super.foo()` --, ist nicht möglich.
- Innerhalb von Konstruktoren, und dort nur als erstes Statement, steht die Surrogatmethode `super (...)` zur Verfügung, durch welche der entsprechende Superklassenkonstruktor aufgerufen werden kann. Wird der Konstruktor der Superklasse nicht explizit aufgerufen, so erfolgt automatisch ein Aufruf des parameterlosen Standardkonstruktors der Superklasse.
- Destruktorenaufrufe werden (analog `C++`) nicht an die Superklasse propagiert.

```

(1)public class ConstDestProp {
(2)    public static void main(String[] args) {
(3)        System.out.println("first creation...");
(4)        new C2();
(5)        System.gc();
(6)
(7)        System.out.println("second creation...");
(8)        new C2(42);
(9)
(10)       System.out.println("third creation...");
(11)       new C2(3.14);
(12)    } //main()
(13)} //class ConstDestProp
(14)
(15)class C1 {
(16)    public C1(int i) {
(17)        System.out.println("constructor of C1 executed with param i="+i);
(18)    } //C2(int)
(19)
(20)    public C1() {
(21)        System.out.println("constructor of C1 executed");
(22)    } //C1()
(23)
(24)    public void finalize() {
(25)        System.out.println("destructor of C1 executed");
(26)    } //finalize()
(27)} //class C1
(28)
(29)class C2 extends C1 {
(30)    public C2() {
(31)        System.out.println("constructor of C2 executed");
(32)    } //constructor
(33)
(34)    public C2(int i) {
(35)        super(i++);
(36)        System.out.println("constructor of C2 executed with param i="+i);
(37)    } //constructor
(38)

```

```

(39)    public C2(double d) {
(40)        System.out.println("constructor of C2 executed with param d="+d);
(41)    } //constructor
(42)
(43)    public void finalize() {
(44)        System.out.println("destructor of C2 executed");
(45)    } //finalize()
(46)} //class C1

```

Beispiel 49: Propagierung von Konstruktoren- und Destrukturaufrufen [ConstDestProp.java](#)

Bildschirmausgabe:

```

$java ConstDestProp
first creation...
constructor of c1 executed
constructor of c2 executed
destructor of c2 executed
second creation...
constructor of c1 executed with param i=42
constructor of c2 executed with param i=43
third creation...
constructor of c1 executed
constructor of c2 executed with param d=3.14

```

Zunächst wird ein Objekt der Klasse C2 über den Aufruf des parameterlosen Konstruktors erzeugt. Vor Ausführung des Konstruktors von C2 wird durch den Übersetzer ein Aufruf an den Superklassenkonstruktor von c1 generiert.

Bei Entfernung des C2-Objektes aus dem Speicher wird dessen Destruktor, nicht jedoch der der Superklasse, aufgerufen.

Die zweite Objekterzeugung nutzt den expliziten parametrisierten Konstruktor C2(int). Innerhalb dieser Methode wird zunächst explizit der Konstruktor identischer Parameterliste der Superklasse explizit per super-Schlüsselwort aufgerufen.

Die dritte Sequenz nutzt den zweiten parametrisierten Konstruktor in C2. In dessen Rumpf ist jedoch kein expliziter Aufruf eines Superklassenkonstruktors plziert. Daher wird automatisch ein Aufruf an den parameterlosen Konstruktor der Superklasse erzeugt.

- Methoden identischer Signatur überschreiben das Pendant der Superklasse.
- Sichtbarkeitseinschränkungen an Methoden und Attribute können in der Subklasse gegenüber der Superklasse erweitert werden.

Beispiel: Superklasse: protected, Subklasse: private ist möglich.

- **Abstrakte Klassen** werden durch Angabe des Schlüsselwortes abstract als nicht instanzierbar gekennzeichnet.
- Anders als in C++ besitzt *jede* Javaklasse eine Superklasse. Ist eine solche nicht explizit angegeben, so setzt der Compiler, außer für die Klasse Object selbst, die API-Klasse [Object](#) an diese Stelle.

Anmerkung: Gewisse generische Möglichkeiten (z.B. Objektsammlungen oder [Collections](#)) werden hierdurch erst eröffnet.

- Die Vererbungssemantik gilt auch für Arrays!
So kann jeder beliebige Array `T[]` in `Object[]` umgewandelt werden (*up cast*).

▲ 2.4.10 Schnittstellen

In C++ nicht explizit präsent. Dort wird die Schnittstellensemantik durch *pure virtual classes* (abstrakte Klasse die nur abstrakte Methoden besitzt) nachgebildet.

Hinweis: Trotz der teilweise weit gefaßten Interpretation des Begriffes *Schnittstelle* wurde diese Übersetzung hier für das originalsprachliche *Interface* verwendet.

Java-Schnittstellen versammeln [Operationen](#), d.h. Methodensignaturen ohne eine Implementierung vorzugeben, sowie Konstante.

Zusätzlich können auch Konstanten definiert werden.

Syntax:

```

InterfaceDeclaration:
    InterfaceModifiersopt interface Identifier

```

ExtendsInterfaces_{opt} InterfaceBody

Auch für Schnittstellen stehen die bekannten Modifier `public`, `protected`, `private`, `abstract`, `static` und `strictfp` zur Verfügung.

Die Sichtbarkeitseinschränkungen sind genauso definiert wie die gleichnamigen Pendanten für Klassen. Konsequenterweise muß ein `public` Interface auch in einer gleichnamigen Quellcodedatei untergebracht werden.

Zwar definiert die Sprachspezifikation (aus Konsistenzgründen zur Definition der Klasse) den **Modifier `abstract`**, dieser ist aber für alle Schnittstellen implizit. Die [Sprachspezifikation](#) rät sogar explizit von seiner (verwirrenden) Verwendung ab.

Weiterführende Information: [Java Language Specification](#)

Charakteristika:

- Verhalten identisch zu abstrakten Klassen (Vererbung, Substitution, nicht instanzisierbarkeit).
- Während des Übersetzerlaufes wird geprüft, ob eine Klasse alle Operationen der Schnittstelle implementiert. Ist dies nicht der Fall, so wird die Deklaration der Klasse als abstrakt erzwungen.
- Auf Ebene der ist Mehrfachvererbung (genauer: Mehrfachimplementierung) zugelassen. Mehrfach geerbte signaturgleiche Operationen werden zu einer einzigen vereinigt.
Verursacht durch die (möglicherweise) intendierte unterschiedliche Semantik der verschiedenen implementierten Schnittstellen kann dies zu ernsthaften Problemen führen!
- Durch Schnittstellenimplementierung erhält die Klasse als zusätzlichen Typ den (Namen) der Schnittstelle.
- Attribute einer Schnittstelle sind implizit `final static`, und damit als Konstanten, deklariert.
- Implementiert eine Klasse eine Schnittstelle, so vererbt sich dies an alle Subklassen.
- Operationen einer Schnittstelle sind implizit als `abstract`, und damit -- innerhalb der Schnittstelle als -- nicht implementierbar, deklariert.
- Schnittstellen können andere Schnittstellen beerben. Auch hier gilt Einfachvererbung.
- Analog zu den [anonymen inneren Klassen](#) können auch anonyme innere Interfaces erstellt werden. Dasselbe gilt für benannte innere Klassen.

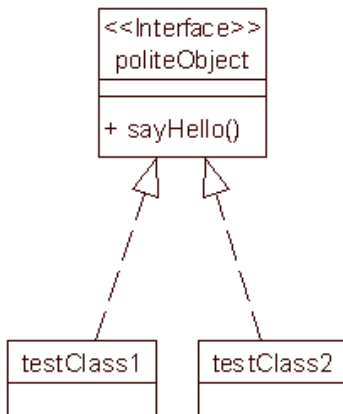


Abbildung 10: UML-Darstellung des Beispiels (click on image to enlarge!)

```

(1)public class Interface1 {
(2)    public static void main(String[] args) {
(3)        TestClass1 tco1 = new TestClass1();
(4)        TestClass2 tco2 = new TestClass2();
(5)        tco1.sayHello();
(6)        tco2.sayHello();
(7)
(8)        if (tco1 instanceof TestClass1)
(9)            System.out.println("tco1 is instance of TestClass1");
(10)
(11)        if (tco1 instanceof PoliteObject)
(12)            System.out.println("tco1 is instance of PoliteObject");
(13)
(14)        PoliteObject po;
(15)        po = tco1;
(16)        po.sayHello();
(17)        po = tco2;
(18)        po.sayHello();
(19)    } //main()
(20)} //class Interface1
(21)
  
```

```

(22)interface PoliteObject {
(23)    public void sayHello();
(24)} //interface PoliteObject
(25)
(26)class TestClass1 implements PoliteObject {
(27)    public void sayHello() {
(28)        System.out.println("Hello World");
(29)    } //sayHello()
(30)} //class TestClass1
(31)
(32)class TestClass2 implements PoliteObject {
(33)    public void sayHello() {
(34)        System.out.println("Guten Tag");
(35)    } //sayHello()
(36)} //class TestClass2

```

Beispiel 50: Verwendung von Schnittstellen [Interface1.java](#)

BildschirmAusgabe:

```

$java Interface1
Hello World
Guten Tag
tco1 is instance of TestClass1
tco1 is instance of politeObject
Hello World
Guten Tag

```

Das Beispiel definiert die Schnittstelle `PoliteObject` welche die Operation `sayHello` anbietet. Die beiden Klassen `TestClass1` und `TestClass2` implementieren jeweils Methoden für die durch die Schnittstelle definierten Operationen.

Der erste Anweisungsblock zeigt die (simple) Ausführung der genannten Methode.

Die darauffolgende Codesequenz hebt den Aspekt der Mehrfachtypisierung (Objekt von Schnittstellen-implementierender Klasse ist sowohl Ausprägung der Klasse selbst (im Beispiel: `TestClass1`, als auch der Schnittstelle (`PoliteObject`)).

Abschließend wird eine Variable vom Typ der Schnittstelle deklariert, und zunächst mit der Referenz auf ein Objekt der Klasse `TestClass1` belegt. Der (statisch typischer mögliche) Aufruf `sayHello` ist auch hier möglich. Gleiche Bedingungen herrschen nach der Zuweisung eines Objektes vom Typ `TestClass2` an dieselbe Variable.

Einige **bekannte Schnittstellenverwendungen**:

- [Comparable](#) -- wird u.a. von allen numerischen Wrappertypen implementiert.
- [Serializable](#) -- dient zur Speicherung (Serialisierung) von Objekten.

2.4.10 Pakete

Pakete stellen eine Sammlung von Klassen und Schnittstellen dar, die Zugriffsschutz und Namensraumverwaltung bietet.

[siehe Java Language Specification](#)

Aus dieser Definition heraus sind die Java-Pakete mit den aus C/C++ bekannten Header- und Includedateien nur schwer vergleichbar. Anders als in C/C++ werden diese externen Quellcodesequenzen nicht physisch in den zu übersetzenden Strom eingebunden, sondern existieren weiter und unabhängig. Technisch stellen Pakete ein eigenständiges Sprachmittel dar. Insbesondere dienen sie nicht zur Separierung von Schnittstellendefinition (.h-Datei) und deren Implementierung.

Syntax einer Paketdefinition:

`package packageName;` am Anfang der Quellcodedatei.

Syntax einer Paketdefinition:

`import packageName.subPackage.subSubPackage... className;`

Hinweis: Im Gegensatz zur C/C++-Präprozessoranweisung `include` stellt die `import`-Definition einen syntaktisch korrekten Javaausdruck dar, der durch ein Semikolon abgeschlossen werden muß.

Paketbenennung: Um Pakete weltweit eindeutig identifizieren und unterscheiden zu können hat sich als Konvention ein an die URL-Notation ([IETF RFC 1738](#)) angelehntes Namensschema

durchgesetzt. Hierbei werden die URL-Komponenten ausgehend von der Toplevel-Domain von rechts nach links angegeben.

Beispiel: Ein Paket `testPackageA` im Namensraum `myCompany.germany.org` würde als `org.germany.myCompany.testPackageA` abgelegt.

Die einzelnen Hierarchieebene der Paketidentifikation werden im JDK durch Kataloge im Dateisystem nachgebildet in denen die Javodateien des entsprechenden Paketes abgelegt werden müssen. So würde die Datei `testPackageA.class` im Katalog `/com/germany/myCompany` plaziert (Durch Substitution der Punkte im vollqualifizierten Klassennamen durch den Verzeichnistrenner `/` ergibt sich der vollqualifizierte Dateipfad.

Das JDK erfordert es, alle Quellcodedateien einer Paketebene gemeinsam zu übersetzen.

Compileraufruf `javac File1.java File2.java File3.java ...`

[Weiterführende Information](#)

Charakteristika:

- Eine Javodatei kann maximal ein Paket beinhalten.
- Jede Quellcodedatei kann beliebig viele Pakete importieren.
- Pakete können geschachtelt werden; d.h. ein Paket kann neben Klassen und Schnittstellen auch weitere Pakete beinhalten.
- Klassen und Schnittstellen in Quellcodedateien ohne explizite einleitende Paketdefinition werden einem unbenannten Standardpaket (*default package*) zugeordnet.
- Alternativ zur Ablage der Paketstruktur im Dateisystem kann der entsprechende Dateibaumast auch zu einer einzigen JAR-Datei (intern ZIP-Komprimierung) zusammengefaßt werden.
- Import kann verschiedengranular angegeben werden:
 - Vollqualifiziert bis auf Klassenebene -- importiert wird (ausschließlich) die spezifizierte Klasse. Der Importvorgang erfolgt erst beim tatsächlichen Zugriff (*import on demand*)
 - Auf beliebiger Ebene endend, mit Stern `*` terminiert -- importiert werden alle Unterpakete einschließlich der dort definierten Klassen.
- Für Pakete gelten besondere [Sichtbarkeitseinschränkungen](#).
Besonders hervorzuheben ist die Zugriffsbehandlung nicht `public` deklarierter Attribute oder Methoden innerhalb eines Paketes. Hier werden alle Elemente desselben Paketes als vertrauenswürdig (engl. *trusted*) angesehen, und erhalten daher auch Zugriff auf als `protected` deklarierte Information.
- Auf Klassen und Schnittstellen importierter Pakete kann ohne Qualifizierung zugegriffen werden.
- Zum Übersetzungszeitpunkt wird die Namenseindeutigkeit der importierten Elemente geprüft. Liegen mehrdeutige Namensdeklarationen vor, so wird dies durch den Compiler gemeldet.

Hinweise:

- Das Paket [java.lang](#) wird durch den Übersetzer automatisch importiert. Daher stehen Methoden wie [System.out.println](#), [System.gc](#) ohne vorhergehende `import`-Anweisung zur Verfügung.
- Inhalte des *Java Extension* Paketes `javax` sind nicht Bestandteil des offiziellen JDKs und stehen daher oftmals nicht auf allen unterstützten Plattformen zur Verfügung.
- Die Umgebungsvariable `CLASSPATH` muß zwingend zumindest Dateisystemkataloge der benutzerdefinierten Pakete umfassen. In älteren JDK-Versionen (`< v1.3`) zusätzlich noch die Systempakete, bzw. die Datei `classes.zip`.

Beispiele aus der Java-API:

- [Übersicht der Pakete](#)
- [java.lang](#), enthält u.a. die [Wrappertypen](#)
- [java.applet](#), Grundpaket der [Applet-Programmierung](#)
- [java.io](#), [Ein-/Ausgabebehandlung](#)
- [java.util](#), u.a. [Collection-API](#)

Beispiel:

Quellcodedateien:

```
(1)package testPackage;
(2)
(3)public class TestClass {
(4)    public static void sayHello() {
(5)        System.out.println("hello from TestClass contained in testPackage");
(6)    } //sayHello()
(7)} //class TestClass
```

Beispiel 51: Klasse `TestClass` innerhalb des Paketes `testPackage` [TestClass.java](#)

```

(1)import testPackage.*;
(2)
(3)public class PackageUser {
(4)    public static void main(String[] args) {
(5)        testClass.sayHello();
(6)    } //main()
(7)} //class PackageUser

```

Beispiel 52: Paketverwendende Datei PackageUser.java [PackageUser.java](#)

Dateiplazierung:

TestClass.java im Verzeichnis testPackage.

packageUser.java im logisch direkt übergeordneten Verzeichnis.

Anmerkung:

Der allgemeine Import aller in testPackage enthaltenen Klassen, Schnittstellen und Pakete könnte auch per `import testPackage.TestClass;` auf die gewünschte Klasse TestClass eingeschränkt werden.

Ebenso würde der vollqualifizierte Methodenaufruf mit `testPackage.TestClass.sayHello();` ohne `import`-Deklaration auskommen.

Statische Imports

Mit der Überarbeitung zur Sprachversion 1.5 wurde durch die *statischen Importe* eine abkürzende Schreibweise für importierte statische Methoden etabliert.

Durch die zusätzliche Angabe des Schlüsselwortes `static` vor der vollqualifizierten Klassenidentifikation stehen als Resultat alle als statisch deklarierten Methoden zum direkten Aufruf, d.h. ohne den Zwang die deklarierende Klasse zu explizieren, zur Verfügung. Semantisch entspricht die neu geschaffene Importvariante jedoch der bisher vorstellten Mimik. Sie ergänzt diese lediglich um eine Schreibweise die syntaktisch an den Aufruf globaler Funktionen in C/C++ erinnert.

Das nachfolgende Beispiel zeigt die Anwendung beim Aufruf der Methode [sin](#) der Standardklasse [Math](#):

```

(1)import static java.lang.Math.*;
(2)
(3)public class SITest {
(4)    public static void main(String[] atrgs) {
(5)        System.out.println("sin(42)="+ sin(42) );
(6)    } //main()
(7)} //class SITest

```

Beispiel 53: Statischer Import [SITest.java](#)

▲ 3 Die Java-Plattform

▲ 3.1 Die Laufzeitumgebung

▲ 3.1.1 Garbage Collection

Wie bereits im [Einführungskapitel](#) angeschnitten, verfügt die Javalauftzeitumgebung über eine Speicherverwaltung automatischer Speicherbereinigung (engl. *garbage collection*) des dynamisch verwalteten Heaps.

Generell kann Speicher durch den Programmierer nur konsumiert, nicht jedoch wieder freigegeben werden. (Zur Erinnerung: Die [explizite Nullzuweisung](#) an eine Objektvariable markiert den dadurch referenzierten Speicherplatz als nicht mehr benötigt, gibt ihn jedoch nicht direkt frei.)

Speicherbereiche werden in Java durch das Java-Schlüsselwort [new](#), für Java-Objekte, bzw. durch die [explizite Definition von primitiven Datentypen](#), reserviert. Konkret geschieht die

Speicherreservierung auf dem Heap ausschließlich durch die Byte-Code-Instruktionen [new](#), [newarray](#), [anewarray](#) und [multianewarray](#).

Explizite Freigabemöglichkeiten, wie durch `delete` in C++, existieren nicht.

Zum Begriff *garbage collection*: Die Wortwahl impliziert, daß nicht mehr benötigter Speicher als Abfall behandelt wird, der wegzuwerfen ist. Jedoch implementiert die automatische Speicherbereinigung nicht das intuitiv damit verknüpfte Bild des weggebens... Vielmehr führt die *garbage collection* ein *Speicher Recycling* durch, in dessen Verlauf wiederverwendbare Speicherbereiche erkannt, und einer neuen Nutzung zugeführt werden.

Unabhängig von der tatsächlichen Implementierung vollzieht sich die Speicherbereinigung in zwei Schritten:

- Identifikation von nicht mehr erreichbaren Speicherobjekten.
- Freigabe der ermittelten Speicherobjekte.

Als zusätzliche Implementierungsrestriktion tritt unter praktischen Gesichtspunkten noch hinzu, daß der Garbage-Collector-Lauf möglichst wenig (im Idealfalle: keinen) zusätzlichen Speicher beansprucht. Nur so kann das funktionieren auch bei knappem Speicher noch gewährleistet werden.

Für die Aufgabenstellung der automatischen Freispeichergewinnung werden daher sog. *mark and sweep*-Algorithmen eingesetzt. Die zugrundeliegende Vorgehensweise kann wie folgt beschrieben werden: während seiner periodischen Durchläufe markiert der Speicherbereinigungsprozeß alle erreichbaren Speicherobjekte (*mark*-Phase). Nach Analyse aller möglichen Zugriffspfade werden die unmarkierten (da unreferenzierten) Speicherobjekte automatisch freigegeben (*sweep*-Phase).

Während der Mark-Phase werden die Speicherreferenzen verändert, daher wird die Programmausführung zu diesem Zeitpunkt unterbrochen. Als Konsequenz hat die Speicherbereinigung meßbare Auswirkung auf die Ausführungszeit. Die Hauptzeit wird jedoch während der Sweep-Phase verbracht, deren Dauer letztlich von der Größe des zur Verfügung stehenden Arbeitsspeichers abhängt.

Die durch das Mark-and-Sweep-Verfahren bedingte zeitweilige Unterbrechung der Programmausführung ist für interaktive- und Realzeitanwendungen denkbar schlecht geeignet. Daher kommen in der Praxis, so auch in Java, zumeist modifizierte -- aber aufwendigere -- Verfahren zum Einsatz, welche einzelne Speicherobjekte dynamisch sperren. Die Modifikationen setzen an der Erkenntnis an, daß sowohl die Markierungs- als auch die Löschphase inkrementell organisiert sind, d.h. sie betreffen nicht die Gesamtheit der speicherresidenten Objekte. Daher eignen sich beide zu einer kontrolliert parallelen Ausführung, die „lediglich“ dafür Sorge tragen muß, daß die aktuell durch den Garbage Collector im Zugriff befindlichen Speicherstrukturen entsprechend gesperrt sind.

Dieses Verfahren ist der naiven Referenzzählung (separate Tabelle enthält Markierung für jedes Objekt, ob Referenzen darauf existieren) deutlich überlegen, da auch zirkulär verkettete nicht erreichbare Speicherstrukturen als unerreichbar erkannt werden können.

Das nachfolgende Beispiel legt sukzessive verschiedene Speicherstrukturen an, im Folgenden wird der Ablauf des *mark and sweep*-Algorithmus verdeutlicht:

Anmerkung: Im Vorgriff auf die Behandlung der [Collection API](#) verwendet die Implementierung der Klasse `node` die Klasse [HashSet](#) und die Schnittstelle [Iterator](#), auf die in [Kapitel 3.2.4](#) näher eingegangen wird.

```
(1)import java.util.HashSet;
(2)import java.util.Iterator;
(3)
(4)public class GCTest4 {
(5)    static int m1;
(6)
(7)    void aMethod(Node paramNode) {
(8)        Node m2 = new Node("n1");
(9)        Node tmpNode;
(10)
(11)        tmpNode = m2.createChild("n11");
(12)        tmpNode.createChild("n111");
(13)        tmpNode = m2.createChild("n12");
(14)        tmpNode.createChild("n121");
(15)        tmpNode.createChild("n122");
(16)        tmpNode.createChild("n123");
(17)}
```



```

(18)         Node m3 = new Node("n2");
(19)         tmpNode = m3.createChild("n21");
(20)         tmpNode.appendChild( m3 );
(21)
(22)         System.out.println("output just for testing reasons...");
(23)         System.out.println("name of parameter Node: "+paramNode.getName() );
(24)         System.out.println("name of Node referenced by m2: "+m2.getName() );
(25)         System.out.println("child's names:");
(26)         m2.getChildsNames();
(27)
(28)         System.out.println("name of Node referenced by m3: "+m3.getName() );
(29)         System.out.println("is n21 child of n2? "+m3.isChild("n21") );
(30)         System.out.println("is n2 child of n21? "+tmpNode.isChild("n2") );
(31)
(32)         tmpNode = new Node("n31");
(33)
(34)         ((tmpNode.createChild("n32")).createChild("n33")).appendChild
(tmpNode);
(35)
(36)         tmpNode = null;
(37)         System.gc();
(38)     } //aMethod()
(39)
(40)     public static void main(String[] args) {
(41)         (new GCTest4()).aMethod(new Node("n4"));
(42)     } //end main()
(43)} //class GCTest4
(44)
(45)class Node {
(46)    HashSet childs = new HashSet();
(47)    String name;
(48)
(49)    public Node(String name) {
(50)        this.name = name;
(51)        System.out.println("created Node object named "+this.name);
(52)    } //Node(String)
(53)
(54)    public void appendChild(Node child) {
(55)        childs.add(child);
(56)    } //appendChild(Node)
(57)
(58)    public Node createChild(String name) {
(59)        Node newChild = new Node( name );
(60)        childs.add(newChild);
(61)        return newChild;
(62)    } //createChild(String)
(63)
(64)    public String getName() {
(65)        return name;
(66)    } //getName()
(67)
(68)    public boolean isChild(String name) {
(69)        Iterator childIt = childs.iterator();
(70)        while (childIt.hasNext()) {
(71)            if ( ((Node) childIt.next()).getName() == name)
(72)                return true;
(73)        } //while
(74)        return false;
(75)    } //isChild(String)
(76)
(77)    public void getChildsNames() {
(78)        Node child;
(79)        Iterator childIt = childs.iterator();
(80)        while (childIt.hasNext()) {
(81)            child = (Node) childIt.next(); //explicit (down) type cast
needed since HashMap contains only Object instances
(82)            System.out.println( child.getName() );
(83)            child.getChildsNames();
(84)        } //while
(85)    } //getChildsNames()
(86)    protected void finalize() {
(87)        System.out.println("Node object named "+name+" freed!");

```



```
(88)    } //finalize()
(89)} //class Node
```

Beispiel 54: verschiedene Speicherstrukturen [GCTest4.java](#)

BildschirmAusgabe:

```
$java GCTest4
created node object named n4
created node object named n1
created node object named n11
created node object named n111
created node object named n12
created node object named n121
created node object named n122
created node object named n123
created node object named n2
created node object named n21
output just for testing reasons...
name of parameter node: n4
name of node referenced by m2: n1
child's names:
n11
n111
n12
n121
n122
n123
name of node referenced by m3: n2
is n21 child of n2? true
is n2 child of n21? true
created node object named n31
created node object named n32
created node object named n33
node object named n31 freed!
node object named n32 freed!
node object named n33 freed!
```

Ablauf des *mark and sweep*-Verfahrens:

Zunächst werden die Referenzen aller im Gültigkeitsbereich der aktuell ausgeführten Methode verfolgt. Im Einzelnen sind dies:

- Übergabe-Argumente
- lokale Variablen
- statische Variablen (der geladenen Klassen)
- Referenzen die durch das [Java Native Interface \(JNI\)](#) registriert sind

Die Menge dieser Speicherreferenzen wird als *root set* bezeichnet.

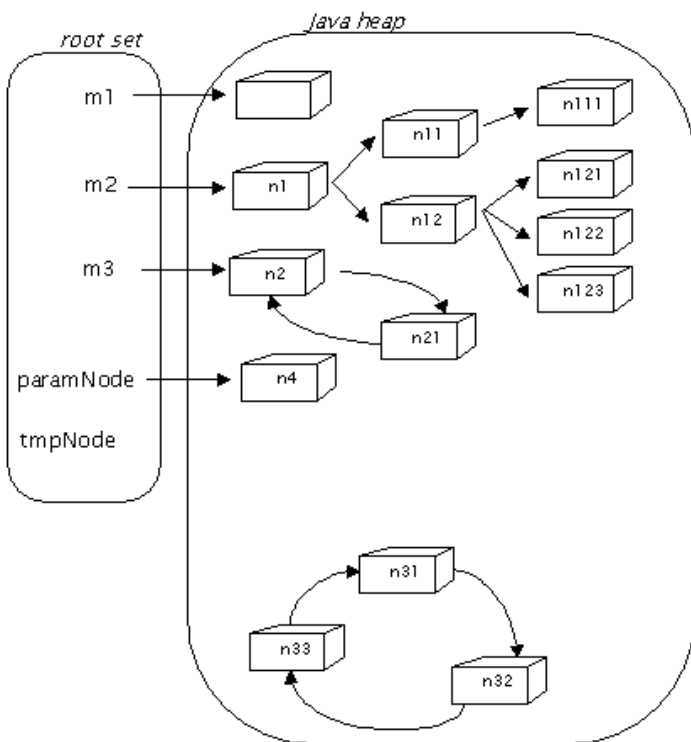


Abbildung 11: Schematische Darstellung der Speicherstrukturen des Beispielprogramms nach null-Setzung von tmpNode, unmittelbar vor Ablauf der Garbage Collectors ([click on image to enlarge!](#))

Mark-Phase:

Der Speicherbereinigungsprozeß durchläuft ausgehend von den Elementen des *root sets*, mit dem Ziel die erreichbaren zu markieren. Die sich zunächst intuitiv aufdrängende Vorgehensweise der rekursiven Baumtraversierung verbietet sich, wenn die [Restriktion](#) daß der Speicherbereinigungsprozeß nur minimale eigene Speicheranforderungen stellt berücksichtigt werden soll. (Darüberhinaus ergibt sich noch ein weit schwerwiegenderes Problem bei zyklischen Strukturen, wie wir in der Folge noch sehen werden...).

Daher wird die verzeigte Speicherstruktur iterativ, unter Abhänderung der Zeigerstruktur, durchlaufen. Da der Rekursionsstack als *Gedächtnis* des genommenen Weges durch den Baum nicht zur Verfügung steht, werden die *Vorwärts*-Zeiger sukzessive zu *Rückkehr*-Zeigern modifiziert. Dies vollzieht sich in zwei Schritten. Zunächst wird der referenzierte Knoten ermittelt und zwischengespeichert. Dann wird dieser besucht und markiert; und die Zeigerrichtung invertiert. Ist ein Blattknoten erreicht und markiert, so werden die veränderten Zeigerstrukturen zur Rückkehr benutzt (sie zeigen jetzt auf den jeweils hierarchisch übergeordneten Knoten). Während des Aufsteigens werden die Zeigerstrukturen nochmals invertiert, d.h. wieder in ihren ursprünglichen Zustand (zurück) versetzt.

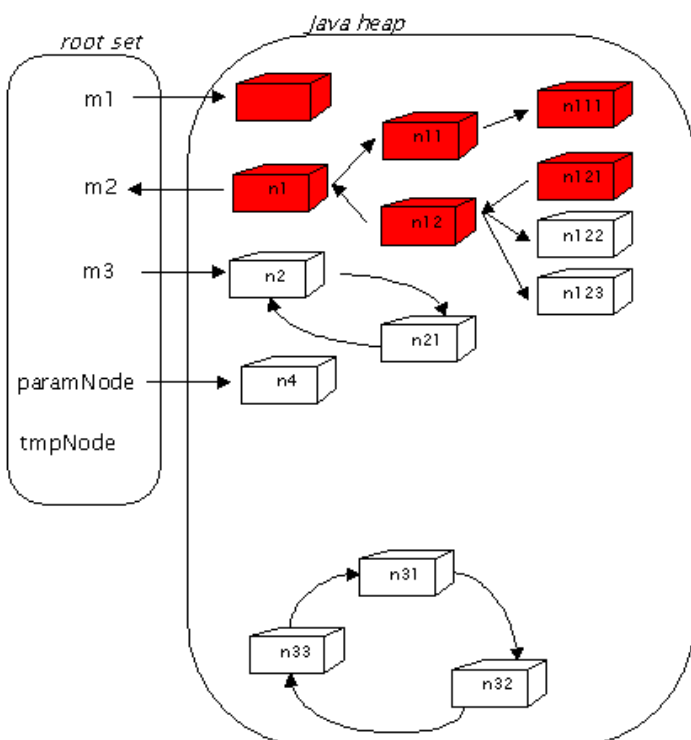


Abbildung 12: Schematische Darstellung der Speicherstrukturen des Beispielprogramms nach Besuch und Markierung einiger Knoten (*click on image to enlarge!*)

Einen Sonderfall, der bei rekursiver Implementierung aufwendig zu behandeln wäre, stellen zirkuläre Strukturen dar. Hierbei handelt es sich allgemein um Zykel beliebiger Länge. Das bedeutet, nach einer gewissen Anzahl von Speicher-knoten existiert ein Verweis (zurück) auf einen bereits traversierten Knoten. Als praktisches Beispiel solcher Strukturen seien zyklisch verkettete Listen angeführt.

Solche Strukturen sind in zweierlei Hinsicht bemerkenswert. Zum Einen, stellen sie hinsichtlich effizienter Traversierung eine Herausforderung dar, zum Anderen, da unabhängige Zykel (siehe n31, n32 und n33 im Beispiel) nicht erreichbare, aber gültig referenzierte Strukturen sind.

Die Graphik zeigt schrittweise die Traversierung und Markierung der einzelnen Knoten eines Zyklus der Länge 2.

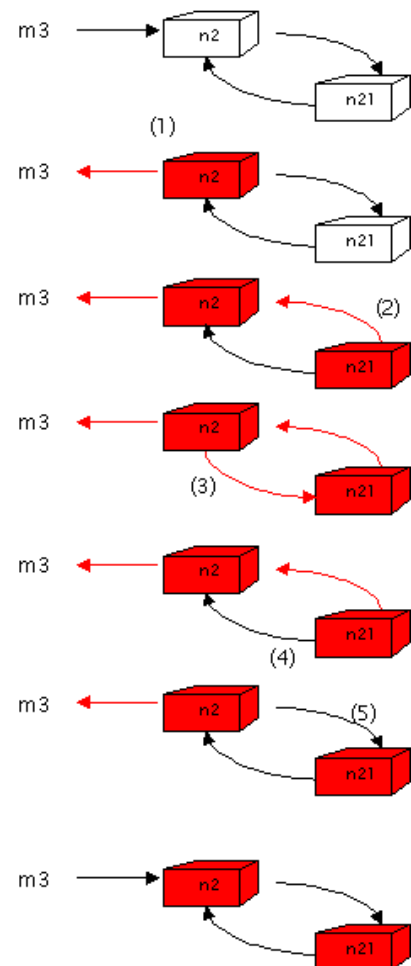


Abbildung 13: Traversierung zyklischer Strukturen (*click on image to enlarge!*)

1. Umkehrung der Referenzierungsrichtung zwischen der Anwendervariable m3 (Element des *root set*) und dem als n2 benannten Speicherobjekt.
n2 wird markiert.
2. Weiternavigation, durch Verfolgung der Referenz von n2 zum Speicherobjekt n21.
Umkehrung der Referenz und Markierung des Knotens n21.
3. Verfolgung der Referenz von n21 zurück zu n2.
Neuer Knoten (n2) ist bereits markiert. (Implizit ist ein Zyklus erkannt worden!)
Damit sind wir zwangsläufig am Ende eines Traversierungsastes angelangt, da auf einen markierten Knoten ausschließlich markierte folgen, sofern die Hierarchie absteigend durchlaufen wird.
4. Verfolgung der Referenz zurück zum Ursprungsknoten.
Anschließend: Invertierung der Verzeigerung; stellt Ursprungszustand wieder her.
5. dto.
6. dto.

Eine Abwandlung des vorhergehenden Falles zyklischer Strukturen stellt die auf n31, n32 und n33 gebildete Struktur dar.

Obwohl auf alle drei erzeugten Objekte gültige Referenzen existieren (n31 zeigt auf n32, n32 auf n33, das wiederum auf n31 verweist) sind die Objekte nach Neuzuweisung (null-Setzung) an tmpNode allesamt nicht mehr erreichbar.

Sweep-Phase:

Sie dominiert den Speicherbereinigungslauf zeitlich. Während die Markierungen vergleichsweise schnell und effizient -- d.h. unter Vermeidung unnötiger Besuchsschritte -- angebracht werden können, wird in der zweiten Phase der gesamte Heap durchlaufen. Dabei wird jedes Speicherobjekt betrachtet, unabhängig davon ob es markiert ist oder nicht.

Nichtmarkierte Speicherobjekte werden in den Freispeicher eingereiht.

Hinweis: Die implementierungsspezifischen Aussagen beziehen sich auf SUNs Java2 (JDK v1.3) Umsetzung.

Technisch ist der Java Garbage Collector eigenständiger niederpriorer Thread innerhalb der [virtuellen Maschine](#) ausgelegt. Situations- und plattformabhängig ist dieser Thread synchron oder asynchron realisiert. Im Falle knappen Speichers, oder expliziter Anforderung durch das Programm, läuft er synchron.

Vor der Freigabe des Speicherplatzes eines Objekts wird dessen [Destruktormethode](#) aufgerufen. Inkrementelle Speicherplatzbereinigung (auf Klassenebene) kann für die virtuelle Maschine der Fa. SUN durch die Kommandozeilenoption `-Xincgc` erzwungen werden.

Der Garbage Collector kann nicht explizit aufgerufen werden. Jedoch erwirkt der Aufruf [System.gc\(\)](#) den Versuch zur Speicherbereinigung. Nach Rückkehr der Methode muß eine Speicherbereinigung nicht zwingend erfolgt sein.

Für Objekte die bereits zur Entfernung aus dem Speicher ausgewählt wurden, jedoch die Abarbeitung ihrer Destruktoren noch aussteht, kann der Destruktlauf mit [System.runFinalization\(\)](#) angestoßen werden.

Die Javaaufzeitumgebung von SUN java erlaubt den Eingriff in die Standardgarbagecollection über folgende Kommandozeilenschalter:

- `verbose:gc`
Schaltet die Protokollierung der Garbage Collector-Aktivitäten auf Stdout ein.
- `Xnoclassgc`
Unterbindet Speicherbereinigung für Klassen.
- `Xincgc`
Aktiviert die inkrementelle Garbage Collection. Hierdurch werden zunächst die (durch Anhalten der Ausführung) spürbaren Speicherbereinigungsläufe zugunsten einer permanenten Ausführung unterbunden. Jedoch geht die Applikationsperformanz um ca. 10% zurück.

```
(1)public class GCTest1 {
(2)    public static void main(String[] args) {
(3)        System.out.println("memory before: "+Runtime.getRuntime().freeMemory
(4)    );
(5)        Test1 t1obj = new Test1();
(6)        System.out.println("memory after: "+Runtime.getRuntime().freeMemory
(7)    );
(8)        t1obj = null;
(9)        System.gc();
(10)       System.out.println("memory after garbage collection: "+ Runtime.
(11)       getRuntime().freeMemory() );
(12)    } //main()
(13)} //class GCTest1
(14)
(15)class Test1 {
(16)    double testArray[] = new double[100];
(17)    public void finalize() {
(18)        System.out.println("object of class Test1 freed");
(19)    } //finalize()
(20)} //class Test1
```

Beispiel 55: Speicherverbrauch vor und nach Garbage Collection [GCTest1.java](#)

BildschirmAusgabe:

```
$java -verbose:gc GCTest1
memory before: 1814608
memory after: 1809824
[Full GC 216K->112K(1984K), 0.0344306 secs]
object of class test1 freed
memory after garbage collection: 1915920
```

Zunächst wird der aktuelle freie Speicher innerhalb der virtuellen Maschine mittels `freeMemory()` abgefragt (genaugenommen liefert die Methode eine Schätzung des freien Speichers in Byte). Das erzeugte Objekt initialisiert einen `double`-Array mit 100 Werten. Hieraus errechnet sich ein minimaler Speicherplatzbedarf von 800 Byte für das Objekt `t1obj`. Der im Beispiel ermittelte Wert von 4520 Byte wird durch weitere Effekte wie Verwaltungsstrukturen und sonstige Laufzeitinformation verursacht.

Nach Nullzuweisung und explizitem Aufruf der Garbage Collectors mit `System.gc()` vergrößert sich der freie Speicher wieder. Er nimmt sogar über den Startwert zu. Dies ist der Freigabe von Speicherobjekten geschuldet, die nicht durch den Anwender, sondern durch die virtuelle Maschine selbst erzeugt wurden. Einen Eindruck der, üblicherweise verdeckt, automatisch geladenen Komponenten liefert der Kommandozeilenschalter `verbose:class` der der Java-Ausführungsumgebung übergeben werden kann.

Die Realisierung des Garbage Collectors von SUN erhebt nicht den Anspruch in jedem Falle Speicherplatz-optimal vorzugehen, d.h. alle potentiell unreferenzierten Objekte zu sofort entdecken und freizugeben. Dies läßt sich in einer Modifikation des obigen Beispiels zeigen:

```
(1)public class GCTest2 {
(2)    public static void main(String[] args) {
(3)        System.out.println("memory before: "+Runtime.getRuntime().freeMemory
(4)    );
(5)        Test1 t1obj = new Test1();
(6)        t1obj = null;
(7)        for (int i=0; i<10; i++) {
(8)            System.gc();
(9)            System.out.println("memory after garbage collection: "+
Runtime.getRuntime().freeMemory() );
(10)        } //for
(11)    } //main()
(12)} //class GCTest2
(13)
(14)class Test1 {
(15)    double testArray[] = new double[100];
(16)    public void finalize() {
(17)        System.out.println("object of class Test1 freed");
(18)    } //finalize()
(19)} //class Test1
```

Beispiel 56: Mehrmaliger Garbage Collectorlauf [GCTest2.java](#)

Bildschirmausgabe:

```
$java -verbose:gc gcTest2
memory before: 1814640
[Full GC 216K->112K(1984K), 0.0336478 secs]
object of class test1 freed
memory after garbage collection: 1915920
[Full GC 113K->112K(1984K), 0.0382289 secs]
memory after garbage collection: 1915952
[Full GC 113K->112K(1984K), 0.0306936 secs]
memory after garbage collection: 1915952
[Full GC 113K->111K(1984K), 0.0392094 secs]
memory after garbage collection: 1917008
[Full GC 112K->111K(1984K), 0.0315576 secs]
memory after garbage collection: 1917008
[Full GC 112K->111K(1984K), 0.0305033 secs]
memory after garbage collection: 1917008
[Full GC 112K->111K(1984K), 0.0313825 secs]
memory after garbage collection: 1917008
[Full GC 112K->111K(1984K), 0.0304195 secs]
memory after garbage collection: 1917008
[Full GC 112K->111K(1984K), 0.0319465 secs]
memory after garbage collection: 1917008
[Full GC 112K->111K(1984K), 0.0361018 secs]
memory after garbage collection: 1917008
```

So wird ein weiterer Speicherblock, trotz der bereits erfolgten Entfernung des Objektes aus dem Speicher (siehe Ausgabe des Destruktors), der Größe 193 Bytes erst durch den vierten Garbage

Collector Lauf freigegeben.

Auswirkungen auf die Programmierung:

A priori hat das Vorhandensein eines automatischen Speicherbereinigungsmechanismus keine Auswirkungen auf die Algorithmenentwicklung oder -umsetzung. Jedoch können Maßnahmen zur expliziten Freigabe nicht mehr benötigter Speicherbereiche durch den Anwendungsprogrammierer (oftmals) unterbleiben, da das Laufzeitsystem sich um die Freigabe nicht mehr erreichbarer Speicherzellen kümmert. Dies kann beispielsweise bei der Implementierung verketteter Datenstrukturen (Listen, Bäume, etc.) hilfreich sein.

Die entstehenden Algorithmen sind jedoch dann nicht mehr adaptionsfrei auf Systeme ohne garbage collection übertragbar.

Abschlußbemerkungen:

- In Einzelfällen kann der berechtigte Wunsch entstehen, Objekte freizugeben, auf die noch Referenzen existieren; beispielsweise wenn ein Objekt vollständig aus dem Speicher entfernt werden soll, obwohl es noch innerhalb von Objektsammlungen (Listen, Arrays, etc.) referenziert wird. Hierfür wurden in Java v1.2 sog. *schwache Referenzen* geschaffen, die es erlauben ausschließlich schwach referenzierte Objekte freizugeben. Implementiert ist dieses Verhalten, das so auch in anderen Programmiersprachen, wie z.B. SmallTalk verwirklicht ist, im Paket [java.lang.ref](#).
- Die technische Umsetzung der automatischen Speicherbereinigung (asynchroner Thread, sub-optimales Freigabeverhalten) bedingt insgesamt ein nichtdeterministisches Anwendungsverhalten hinsichtlich Speicherplatzbedarf der Applikation, gepaart mit einem nichtdeterministisches Laufzeitverhalten durch nicht-planbare und -vorhersehbare Aktivität des Garbage Collectors. Dies läßt die Verwendung von Java, in der Standardversion, für harte Realzeitanwendungen nicht zu. Daher haben sich in der zwischenzeit einige Aktivitäten zur Modifikation von Java für ebendiese Anwendungsdomänen formiert.

▲ 3.1.2 Virtuelle Maschine

Kern der oft apostrophierten [Plattformunabhängigkeit](#) der Programmiersprache Java ist die Generierung eines generischen Zwischenformates -- des Byte-Codes. Dieser wird von einer plattformabhängig implementierten Programmeinheit, der *Java Virtual Machine* (Abk. JVM) interpretativ zur Ausführung gebracht.

Jede Java-Applikation wird auf einer eigenen virtuellen Maschine zur Ausführung gebracht. Dies garantiert eine größtmögliche Abschottung, mit dem Ziel maximierter Sicherheit, der möglicherweise gleichzeitig auf einer realen Maschine ausgeführten Java-Programme voneinander. Das Konzept der virtuellen Maschine, die als Programm auf einer realen abläuft, ermöglicht eine vergleichsweise einfache und schnelle Portierbarkeit auf neue Zielumgebungen, da lediglich die virtuelle Maschine an die veränderte reale Maschine angepaßt werden muß.

Der Gedanke virtueller Maschinen, die generischen Zwischencode -- oftmals auch als *P-Code* bezeichnet -- ausführen, ist nicht neu. Bereits USCD Pascal, E-BASIC und die verschiedenen SmallTalk-Implementierungen, setzt diesen praktisch um.

Die Realisierung der Befehlsfolgen (Opcodes) innerhalb der virtuellen Maschine von Java ähnelt teilweise frappant der Architektur der für die Züricher Pascal-Implementierung entwickelten (abstrakten) P-Maschine.

Ein Beispiel einer virtuellen Maschine ist [java](#), die Bestandteil des Java-Development Toolkits von SUN ist.

Bekannte andere virtuelle Maschinen sind: [Kaffe](#) oder auch [IBMs Jikes-Implementierung](#). [Wie bereits bekannt](#) wird eine Java-Applikation durch den Aufruf `java`, gefolgt vom Namen der Startklasse und etwaiger Kommandozeilenparameter ausgeführt. Technisch gesehen bewirkt der Aufruf zunächst die Erzeugung einer neuen Instanz der virtuellen Maschine, auf welcher die Programm-Abarbeitung mit der [main-Methode](#) der Startklasse begonnen wird.

Eine Instanz einer virtuellen Maschine existiert, solange Programmfäden (engl. *Threads*) (genaugenommen: non-daemon Threads) ausgeführt werden, bzw. die virtuelle Maschine explizit beendet wird (mit dem API-Aufruf [System.exit\(\)](#)) oder ein Fehler auftritt.

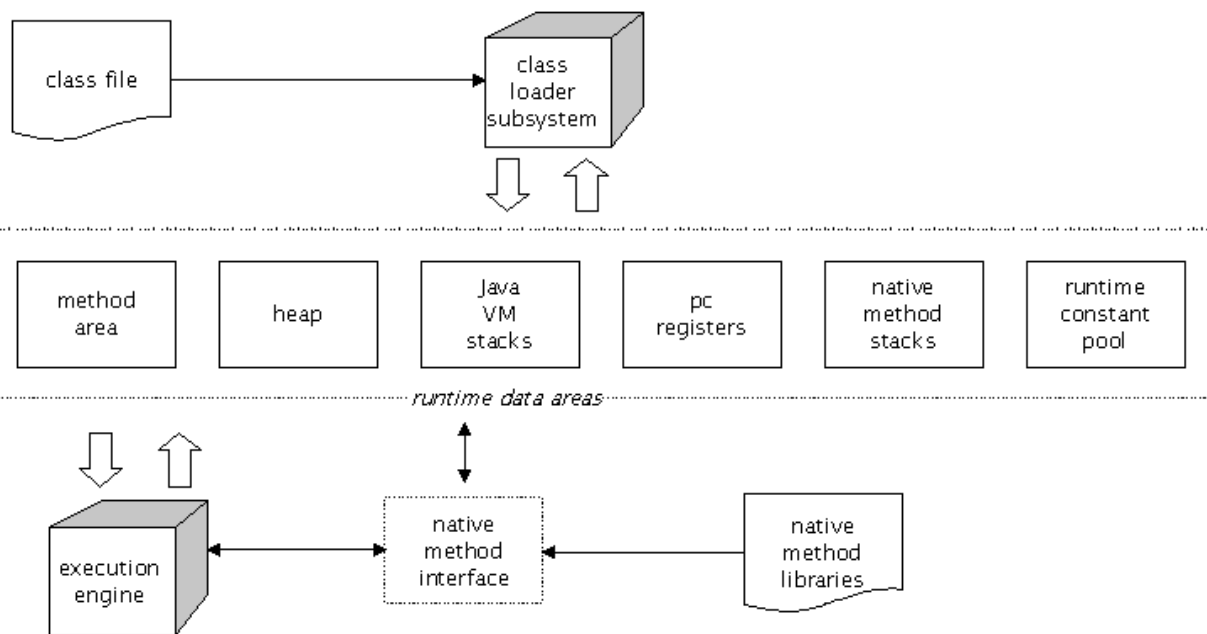


Abbildung 14: Architektur der virtuellen Maschine (nach: Venners, B.: Inside the Java 2 Virtual Machine, chap. 5) (click on image to enlarge!)

Die wesentlichen Bestandteile der JVM sind:

- **method area**
Einmal vorhanden je virtueller Maschine, wird gemeinsam von allen Threads benutzt; enthält klassenspezifische Daten (Typinformation).
- **heap**
Einmal vorhanden je virtueller Maschine, wird gemeinsam von allen Threads benutzt; enthält die dynamisch erzeugten Objekte.
- **Java VM stacks**
Threadspezifisch. Enthält Zustand von nicht-nativen Methodenaufrufen, deren lokale Variablen, Übergabeparameter, den möglichen Rückgabewert sowie Methoden-interne Berechnungsergebnisse. ([vgl. JVM-Spezifikation](#))
- **pc registers**
Threadspezifisch; enthält für jeden Zeitpunkt der Ausführung einer nicht-nativen Methode die Adresse der derzeit ausgeführten Instruktion. ([vgl. JVM-Spezifikation](#)). Während der Abarbeitung nativer Methoden ist der Wert auf undefined gesetzt. Konkrete Größe des virtuellen pc-Registers hängt von der Adresslänge der realen Plattform ab.
- **native method stack**
Implementierungsspezifischer Speicherbereich zur Behandlung von nativen Methodenaufrufen.
- **runtime constant pool**
Klassen- oder Schnittstellenspezifisch. Enthält verschiedene Konstanten, die zur Übersetzungszeit feststehen und in der constant_pool Tabelle der class-Datei abgelegt sind. Die Funktion dieses Bereichs ähnelt dem einer konventionellen Symboltabelle. ([vgl. JVM-Spezifikation](#))

Die Java-Stacks sind in *stack frames* organisiert. Jedem Methodenaufruf ist ein [Stack-Frame](#) zugeordnet, der beim Aufruf erzeugt (push), und beim Verlassen (pop) vom Stack entnommen wird. Innerhalb eines Frames befindet sich

- Array der lokalen Variablen
- Operanden-Stack
Wichtigste Datenstruktur innerhalb der virtuellen Maschine. Der Operanden-Stack enthält alle Eingangs- und Ausgangsdaten der verschiedenen Berechnungen. Seine Einträge sind typisierten in 32-Bit Worten organisiert; die korrekte Typisierung der Eingangsoperanden wird von jedem Opcode geprüft.
- Referenz auf runtime constant pool
- implementierungsspezifische- und debugging-Information

Den für den Anwendungsentwickler offensichtlichsten Bestandteil der virtuellen Maschine, bilden jedoch die JVM-Instruktionen -- die Maschinensprache der JVM.

Der [Befehlssatz der JVM](#) umfaßt ausschließlich 1 Byte lange Opcodes.

Die JVM ist generell **stack-orientiert**; da bedeutet: Quell- und Zieloperanden der meisten Operationen werden vom Stack entnommen, und das Ergebnis dort abgelegt. Insbesondere existieren, abgesehen von vier Verwaltungsspeicherplätzen je Ausführungs-Thread, keine virtuellen Prozessorregister, um die Implementierungsanforderungen an die reale Plattform zu minimieren.

Als threadlokale Register stehen zur Verfügung:

- [pc -- program counter](#)

Enthält die Adresse der gerade ausgeführten Instruktion.

Handelt es sich um eine native-Methode, die in einer anderen Programmiersprache ausgeführt wird, so ist der Wert *undefined*.

- [optop](#)

Verweist auf die Spitze des [Operanden Stacks](#).

- [frame](#)

Verweist auf die Ausführungsumgebung der derzeit aktiven Methode.

- [vars](#)

Ein Zeiger auf die erste lokale Variable der aktuellen Methode.

Die Adresslänge innerhalb der JVM ist auf vier Byte (32 Bit) fixiert. Hieraus ergibt sich ein (theoretischer) Adressraum von 4 GB.

Die initiale und maximale Ausdehnung des Heaps kann durch die Kommandozeilenschalter Xms bzw. Xmx gesteuert werden (Beispiel: `java -Xms350M -Xmx500M HelloWorld` führt das [HelloWorld-Beispiel aus Kapitel 1](#) mit einer anfänglichen Speicherausstattung von 350 MB aus, die im Verlaufe des Programmablaufs auf höchstens 500 MB anwachsen kann.)

Das **Typsystem der JVM** lehnt sich eng an das der Hochsprache an. ([Zur Erinnerung: primitive Typen in Java](#))

Zusätzlich erweitert es die Primitivtypen um einen Adresstypen `returnAddress` und führt explizite Referenztypen auf die verschiedenen high-level Typen (Klassen, Schnittstellen, Arrays) ein.

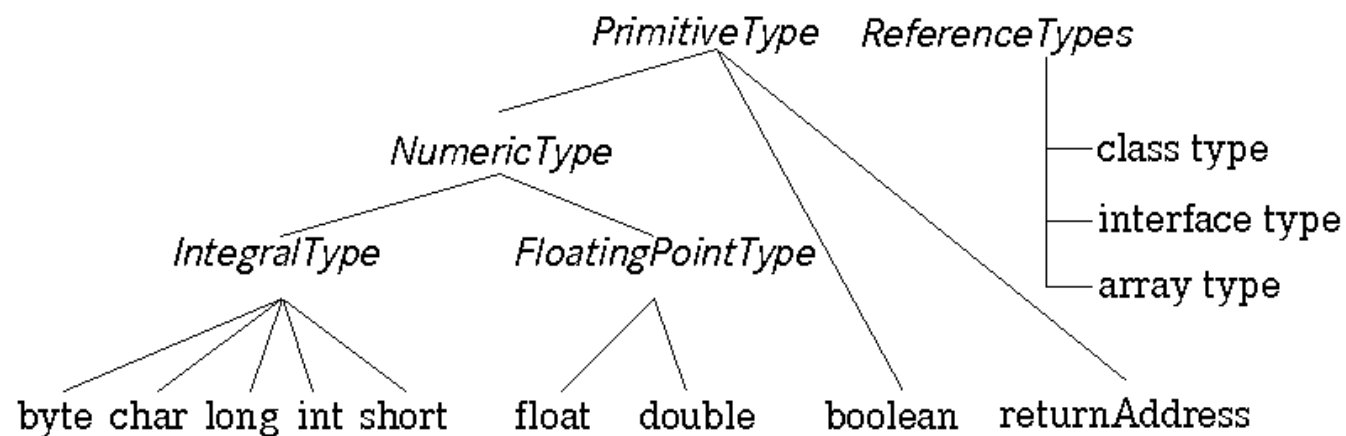


Abbildung 15: Typsystem der virtuellen Maschine (click on image to enlarge!)

Datentypen der JVM:

- `byte`
Vorzeichenbehaftete 8-Bit Zahl in Zweierkomplementdarstellung.
- `short`
Vorzeichenbehaftete 16-Bit Zahl in Zweierkomplementdarstellung.
- `int`
Vorzeichenbehaftete 32-Bit Zahl in Zweierkomplementdarstellung.
- `long`
Vorzeichenbehaftete 64-Bit Zahl in Zweierkomplementdarstellung.
- `char`
Vorzeichenbehaftete 16-Bit Zahl zur Darstellung von Unicode-Zeichen.
- `float`
32-Bit Binärdarstellung einer Fließkommazahl gemäß [IEEE 754-1985](#).
- `double`
64-Bit Binärdarstellung einer Fließkommazahl gemäß [IEEE 754-1985](#).
- `boolean`
Wird zwar durch die JVM definiert, jedoch existieren keine Opcodes, die Parameter dieses Typs erwarten. Der Compiler setzt Anweisungen die `boolean`-Typen enthalten als Operationen auf `int`-Typen um.
- `returnAddress`
Für den Programmierer nicht zugänglicher Typ, der Sprungadressen aufnimmt.
- Referenztypen
Verweisen auf Klassen, Arrays oder Schnittstellen.
Der Wert kann auch `null` sein, wobei die JVM keine konkrete Darstellung dieses Wertes unterstellt.

Jede Bytecode-Instruktion besteht zunächst aus ihrem Opcode, optional gefolgt von den benötigten Operanden. Diese stehen jedoch nicht für sich, sondern sind eingebettet in den organisatorischen Rahmen der `class`-Datei, deren [Format](#) im Anschluß vorgestellt wird.

Die verschiedenen Maschineninstruktionen lassen sich in Klassen einteilen:

Instruktionsfamilie	Vertreter	Anmerkung
Zugriff auf lokale Variablen	ret	Rücksprung aus Subroutine, wird in der Implementierung von finally benutzt
	aload	lädt Referenz von spezifisch indizierter Position auf den Operanden-Stack
	aload <n>	lädt Referenz auf Operanden-Stack (Index im Opcode explizit angegeben)
	astore	legt Referenz von spezifisch indizierter Position in lokaler Variable auf Operanden-Stack ab
	astore <n>	legt Referenz in lokaler Variable auf Operanden-Stack ab (Index wird im Opcode explizit angegeben)
	dload	lädt double-Wert aus lokaler Variable auf den Operanden-Stack
	dstore	lädt double-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
	float	lädt float-Wert aus lokaler Variable auf den Operanden-Stack
	fstore	lädt float-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
	iload	lädt int auf Operanden-Stack (Index im Opcode explizit angegeben)
	iload <n>	lädt int-Wert aus lokaler Variable auf den Operanden-Stack
	istore	legt int von spezifisch indizierter Position in lokaler Variable auf Operanden-Stack ab
	istore <n>	lädt int-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
	lload	lädt long-Wert aus lokaler Variable auf den Operanden-Stack
	lstore	lädt long-Wert vom Operanden-Stack und legt ihn in lokaler Variable ab
	iinc	Inkrementiert lokale Variable um fixe int-Zahl
Explizite Modifikation des Operanden-Stack	bipush	Ablegen eines byte-Wertes auf dem Operanden-Stack
	sipush	Ablegen eines short-Wertes auf dem Operanden-Stack
	pop	Entnimmt (de facto: löscht) obersten Operanden-Stack-Eintrag.
	pop2	Entnimmt (de facto: löscht) obersten beiden Operanden-Stack-Einträge.
	swap	Tauscht die beiden obersten Operanden-Stack-Einträge aus.
	ldc	Ablegen eines Elements (referenziert über 16-Bit Index) des runtime constant pools auf dem Operanden-Stack
	ldc_w	Ablegen eines Elements (referenziert über 32-Bit Index) des runtime constant pools auf dem Operanden-Stack
	aconst_null	Legt null auf dem Operanden-Stack ab.
	dconst <d>	Legt double Konstante auf dem Operanden-Stack ab. dconst_0 die Konstante 0.0, bzw. dconst_1 den Wert 1.0.
	fconst <f>	Legt float Konstante auf dem Operanden-Stack ab. fconst_0 die Konstante 0.0, bzw. fconst_1 den Wert 1.0.

	<u>iconst <i></u>	Legt int Konstante auf dem Operanden-Stack ab. Es existieren Opcodes für folgende Konstanten: iconst_m1 -- -1; iconst_0 -- 0; iconst_1 -- 1; iconst_2 -- 2; iconst_3 -- 3; iconst_4 -- 4; iconst_5 --5.
	<u>dup</u>	Dupliziert oberstes Element des Operanden-Stacks. <u>dup_x1</u> legt das neue Element als zweitunterstes auf dem Stack ab. <u>dup_x2</u> legt das neue Element, abhängig vom Stack-Inhalt, als zweit- oder drittunterstes auf dem Stack ab.
	<u>dup2</u>	Dupliziert die beiden obersten Elemente des Operanden-Stacks. <u>dup2_x1</u> legt die neuen Elemente als zweitunterstes und folgendes auf dem Stack ab. <u>dup2_x2</u> legt die neuen Elemente, abhängig vom Stack-Inhalt, als zweit- oder drittunterstes und folgendes auf dem Stack ab.
	<u>lconst <l></u>	Legt long Konstante auf dem Operanden-Stack ab. Es existieren Opcodes für folgende Konstanten: iconst_0 -- 0; iconst_1 -- 1.
Kontrollfluß	<u>goto</u>	Bedingungsloser Sprung innerhalb derselben Methode. Der anzugebende <i>branch offset</i> ist auf 16-Bit fixiert, woraus sich ableiten läßt, daß das Opcodesegment einer Methode niemals (in der JVM-Version 1.3) die Größe von 64KByte überschreiten darf. Näheres zu den Einschränkungen der virtuellen Maschine findet sich in im Abschnitt 4.10 der JVM-Spezifikation , sowie in der Diskussion des Class-File Formats .
	<u>goto_w</u>	Bedingungsloser Sprung innerhalb derselben Methode (mit 32-Bit Offset)
	<u>if_acmp<cond></u>	Bedingter Sprung im Falle der Gültigkeit der Bedingung. Die zu vergleichenden Operanden werden als Referenzen übergeben. Als Bedingungen stehen Gleichheit (eq) und Ungleichheit (ne) zur Verfügung.
	<u>if_icmp<cond></u>	Bedingter Sprung im Falle der Gültigkeit der Bedingung. Die zu vergleichenden Operanden werden als int-Werte übergeben. Als Bedingungen stehen zur Verfügung: Gleichheit (eq), Ungleichheit (ne), Kleiner (lt), Kleiner oder Gleich (le), Größer (gt) und Größer oder Gleich (ge).
	<u>if<cond></u>	Bedingter Sprung, nach Vergleich des obersten Elements des Operanden-Stacks mit Null Für spezifische Vergleiche stehen folgende Opcodes zur Verfügung: Gleichheit (ifeq), Ungleichheit (ifne), Kleiner (iflt), Kleiner oder Gleich (ifle), Größer (ifgt), Größer oder Gleich (ifge).
	<u>ifnonnull</u>	Bedingter Sprung -- im Falle der Ungleichheit -- nach Vergleich der auf dem Operanden-Stack befindlichen Referenz mit Null
	<u>ifnull</u>	Bedingter Sprung -- im Falle der Gleichheit -- nach Vergleich der auf dem Operanden-Stack befindlichen Referenz mit Null
	<u>jsr</u>	Unbedingter Sprung, unter Sicherung der Rücksprungadresse auf dem Operanden-Stack, zu 16-Bit Adresse.

	<u>jsr_w</u>	Unbedingter Sprung, unter Sicherung der Rücksprungadresse auf dem Operanden-Stack, zu 32-Bit Adresse.
	<u>lookupswitch</u>	Zugriff auf Sprungtabelle per Schlüssel und anschließende Verzweigung. Benutzt zur Implementierung des <u>switch</u> -Konstrukts
	<u>tableswitch</u>	Indexbasierter Zugriff auf Sprungtabelle und anschließende Verzweigung.
Operationen auf Klassen und Objekten	<u>anewarray</u>	Array-Erzeugung an definierter Stelle im <u>Laufzeit-Konstanten-Pool</u> .
	<u>checkcast</u>	Typkompatibilitätsprüfung (<u>siehe Beispiel</u>)
	<u>instanceof</u>	Prüft ob Objekt gegebenen Typ hat (d.h. Ausprägung der Klasse -- oder einer Subklasse -- ist; das Interface implementiert; Array-Kompatibel ist)
	<u>new</u>	Erzeugt neues Objekt <i>Hinweis:</i> Die Punktnotation zur Trennung der Pakethierarchien wird hier durch Slash „/“ ersetzt.
Methodenausführung	<u>invokespecial</u>	Ruft <u>Instanzenmethode</u> auf; mit besonderer Behandlung bestimmter Umstände. <i>Hinweis:</i> Diese Instruktion wurde umbenannt, frühere JDK-Versionen benutzen <code>invokeonvirtual</code> .
	<u>invokestatic</u>	Ruft statische <u>Klassenmethode</u> auf.
	<u>invokevirtual</u>	Ruft Instanzenmethode auf.
	<u>invokeinterface</u>	Ruft Schnittstellenmethode auf.
	<u>areturn</u>	Retourniert Referenz auf Speicherobjekt nach Methodenausführung.
	<u>dreturn</u>	Retourniert double-Wert nach Methodenausführung.
	<u>freturn</u>	Retourniert float-Wert nach Methodenausführung.
	<u>ireturn</u>	Retourniert int-Wert nach Methodenausführung.
	<u>lreturn</u>	Retourniert long-Wert nach Methodenausführung.
	<u>return</u>	Retourniert nach Methodenausführung ohne Rückgabewert (void-Methode).
Attributzugriff	<u>getfield</u>	Legt Attributinhalt eines Objekts auf Operanden-Stack ab. Die Klasse des Objekts, auf das der Zugriff erfolgen soll, wird über einen 16-Bit Offset auf dem <u>runtime constant pool</u> adressiert. Auch hier wird wieder die Limitierung der virtuellen Maschine auf 2 ¹⁶ Klassen deutlich.
	<u>getstatic</u>	Legt Attributinhalt eines statischen Klassenattributes auf dem Operanden-Stack ab.
	<u>putfield</u>	Setzt Wert eines Attributs.
	<u>putstatic</u>	Setzt Wert eines statischen Klassenattributes.
Operationen auf Arrays	<u>newarray</u>	Erzeugt einen neuen Array, und definiert die Komponententypen als einen der <u>Primitvtypen</u> . Die Implementierung von SUN verwendet für den Wahrheitswert je acht Bit. Anderen Umsetzungen ist es explizit Freigestellt hier mit optimierteren Speicherstrukturen zu operieren.
	<u>anewarray</u>	Erzeugt einen neuen Array von Referenztypen zur Aufnahme beliebiger Objekte.
	<u>multianewarray</u>	Erzeugt einen mehrdimensionalen Array.

	<u>aload</u>	Lädt Referenz von Arrayposition.
	<u>astore</u>	Speicher Objekt an spezifischer Arrayposition.
	<u>arraylength</u>	Liefert Elementanzahl (Kardinalität) eines Arrays.
	<u>baload</u>	Lädt byte oder boolean aus Arrayposition.
	<u>bastore</u>	Legt byte oder boolean an Arrayposition ab.
	<u>saload</u>	Lädt short aus Arrayposition.
	<u>sastore</u>	Legt short an Arrayposition ab.
	<u>caload</u>	Lädt char aus Arrayposition.
	<u>castore</u>	Legt char an Arrayposition ab.
	<u>daload</u>	Lädt double aus Arrayposition.
	<u>dastore</u>	Legt double an Arrayposition ab.
	<u>faload</u>	Lädt float aus Arrayposition.
	<u>fastore</u>	Legt float an Arrayposition ab.
	<u>iaload</u>	Lädt int aus Arrayposition.
	<u>iastore</u>	Legt int an Arrayposition ab.
	<u>laload</u>	Lädt long aus Arrayposition.
	<u>lastore</u>	Legt long an Arrayposition ab.
Konvertierungen	<u>i2b</u>	Konvertiert int zu byte.
	<u>i2c</u>	Konvertiert int zu char.
	<u>i2d</u>	Konvertiert int zu double.
	<u>i2f</u>	Konvertiert int zu float.
	<u>i2l</u>	Konvertiert int zu long.
	<u>i2s</u>	Konvertiert int zu short.
	<u>d2f</u>	Konvertiert double zu float.
	<u>l2d</u>	Konvertiert long zu double.
	<u>l2f</u>	Konvertiert long zu float.
	<u>l2i</u>	Konvertiert long zu int.
	<u>d2f</u>	Konvertiert double zu float.
	<u>f2d</u>	Konvertiert float zu double.
	<u>f2i</u>	Konvertiert float zu int.
	<u>f2l</u>	Konvertiert float zu long.
	<u>d2f</u>	Konvertiert double zu float.
	<u>d2f</u>	Konvertiert double zu float.
	<u>d2i</u>	Konvertiert double zu int.
	<u>d2l</u>	Konvertiert double zu long.
Arithmetische Operationen		
Operanden vom Stack und Ergebnisablage wieder darauf	<u>dadd</u>	Addiert zwei double-Werte.
	<u>dsub</u>	Subtrahiert zwei double-Werte.
	<u>ddiv</u>	Dividiert zwei double-Werte.
	<u>dmul</u>	Multipliziert zwei double-Werte.
	<u>dneg</u>	Negiert double-Wert durch Zweierkomplementbildung.

<u>drem</u>	Divisionsrest bei Division zweier double-Werte.
<u>dcmp<op></u>	Vergleicht zwei double Werte. Ist einer der beiden Operanden NaN, so legt dcmpg1, dcmpl hingegen -1 als Ergebnis auf dem Stack ab.
<u>fadd</u>	Addiert zwei float-Werte.
<u>fsub</u>	Subtrahiert zwei float-Werte.
<u>fmul</u>	Multipliziert zwei float-Werte.
<u>fdiV</u>	Dividiert zwei float-Werte.
<u>iadd</u>	Addiert zwei int-Werte.
<u>isub</u>	Subtrahiert zwei int-Werte.
<u>imul</u>	Multipliziert zwei int-Werte.
<u>idiV</u>	Dividiert zwei int-Werte.
<u>iand</u>	Boole'sche UND-Verknüpfung zweier int-Werte.
<u>ior</u>	Boole'sche ODER-Verknüpfung zweier int-Werte.
<u>ixor</u>	Exklusive Boole'sche ODER-Verknüpfung zweier int-Werte.
<u>ineg</u>	Negiert int-Wert durch Zweierkomplementbildung.
<u>irem</u>	Divisionsrest bei Division zweier int-Werte.
<u>ishl</u>	Linksshift eines int-Wertes.
<u>ishr</u>	Rechtsshift eines int-Wertes.
<u>iushr</u>	Rechtsshift eines int-Wertes unter Nulleinfügung und Vorzeichenlösung.
<u>fneg</u>	Negiert float-Wert durch Zweierkomplementbildung.
<u>frem</u>	Divisionsrest bei Division zweier float-Werte.
<u>fcmp<op></u>	Vergleicht zwei floatWerte. Ist einer der beiden Operanden NaN, so legt fcmpg1, fcmpl hingegen -1 als Ergebnis auf dem Stack ab.
<u>ladd</u>	Addiert zwei long-Werte.
<u>land</u>	Boole'sche UND-Verknüpfung zweier long-Werte.
<u>ldcmp</u>	Vergleicht zwei longWerte.
<u>ldiV</u>	Dividiert zwei long-Werte.
<u>lmul</u>	Multipliziert zwei long-Werte.
<u>lneg</u>	Negiert long-Wert durch Zweierkomplementbildung.
<u>lrem</u>	Divisionsrest bei Division zweier long-Werte.
<u>lor</u>	Boole'sche ODER-Verknüpfung zweier long-Werte.
<u>lshl</u>	Linksshift eines long-Wertes.
<u>lshr</u>	Rechtsshift eines long-Wertes.
<u>lsub</u>	Subtrahiert zwei long-Werte.
<u>lushr</u>	Rechtsshift eines long-Wertes unter Nulleinfügung und Vorzeichenlösung.
<u>lxor</u>	Exklusive Boole'sche ODER-Verknüpfung zweier long-Werte.
Ausnahmebehandlung	<u>athrow</u> Wirft eine Exception.
Synchronisation -- Monitoroperationen	<u>monitorenter</u> Der Zugriff auf jedes Objekt wird durch einen Monitor synchronisiert. Die Anweisung sperrt ein Objekt.

[monitorexit](#)

Gibt ein gesperrtes Objekt frei.

Sonstige Opcodes

[nop](#)Buchstäblich: *no operation*. Bewirkt nichts; keine Operatoren, keine Änderungen am Operanden-Stack.

Die beiden Opcodes mit den Ordnungsnummern 254 und 255 (0xfe und 0xff, mnemonic `impdep1` und `impdep2`) sind durch SUN als reserviert gekennzeichnet. Sie können von durch den Hersteller der virtuellen Maschine mit eigendefinierter Funktionalität implementiert werden. Darüberhinaus ist mit dem Opcode 202 (mnemonic `breakpoint`) ein Einstiegspunkt für Debugger definiert.

Alle Opcodes sind mit einem Byte codiert. Hieraus ergibt 256 als maximaler Befehlsumfang der virtuellen Maschine. Zur Verringerung der notwendigen verschiedenen Befehle sind nicht alle Opcodes für alle [Typen der JVM](#) implementiert. Üblicherweise existieren nur Opcodes für `int`, `float`, `long` und `double` sowie die Referenzen. Für alle anderen Typen stehen Konvertierungsmöglichkeiten in die genannten zur Verfügung.

Opcode	byte	short	int	long	float	double	char	Referenz
...ipush	bipush	sipush						
...const			iconst	lconst	fconst	dconst		aconst
...load			iload	lload	fload	dload		aload
...store			istore	lstore	fstore	dstore		astore
...inc			iinc					
...aload	baload	saload	iaload	laload	faload	daload	caload	aaload
...astore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
...add			iadd	ladd	fadd	dadd		
...sub			isubb	lsub	fsub	dsub		
...mul			imul	lmul	fmul	dmul		
...div			idiv	ldiv	fdiv	ddiv		
...rem			irem	lrem	frem	drem		
...neg			ineg	lneg	fneg	dneg		
...shl			ishl	lshl				
...shr			ishr	lshr				
...ushr			iushr	lushr				
...and			iand	land				
...or			ior	lor				
...xor			ixor	lxor				
i2...	i2b	i2s		i2l	i2f	i2d		
l2...			l2i		l2f	l2d		
f2...			f2i	f2l	f2d			
...cmp				lcmp				
...cmpl				fcmpl	dcmpl			
...cmpg				fcmpg	dcmpg			
if_...cmpcond			if_icmpcond					if_acmpcond
...return			ireturn	lreturn	freturn	dreturn		areturn

Treten bei der Ausführung der Opcodes Ausnahmen auf, so werden durch die virtuelle Maschine [Laufzeit-Ausnahmeereignisse](#) (engl. *runtime exception*) generiert. [Wie bekannt](#) werden Ausnahmen dieser Kategorie (üblicherweise) nicht aufgefangen und behandelt.

So wird die [ClassCastException](#) im [Beispiel aus Kapitel 2](#) durch die versuchte explizite Typumwandlung ausgelöst.

Der erzeugte Bytecode für diese Anweisung lautet:

```
aload_3
checkcast 2
```

aload_3 lädt die Referenz auf eine lokale Variable auf den Operanden-Stack. Die lokale Variable 3 entspricht im Beispiel myC11.

checkcast testet ob die auf dem Operanden-Stack befindliche Referenz kompatibel zum übergebenen Typen (hier die 2 als Referenz auf die zweite geladene Klasse; benannt mit C2) ist. Im Falle der Nichtkompatibilität wird durch die virtuelle Maschine eine `ClassCastException` erzeugt.

Ein **umfangreicheres Beispiel**:

Die Methode computeSum...

```
int computeSum(int n) {
    int res = 1;
    for (int i=2; i<=n; i++)
        res += i;
    return res;
}
```

...wird in folgenden Byte-Code übersetzt:

```
0 iconst_1
1 istore_2
2 iconst_2
3 istore_3
4 goto 14
7 iload_2
8 iload_3
9 iadd
10 istore_2
11 iinc 3 1
14 iload_3
15 iload_1
16 if_icmple 7
19 iload_2
20 ireturn
```

Beispiel aus: Kopp, M.: Die virtuelle Java Maschine (2)

Die ersten vier Bytecodebefehle besitzen alle keine Parameter und initialisieren die lokale Variable 2 (entspricht der Java Variablen res) mit 1 und die lokale Variable 3 (entspricht der Java Variablen i) mit 2. Anschließend wird an die Stelle 14 gesprungen, an der der Vergleich von i und dem Parameter (gespeichert in der lokalen Variablen 1) durchgeführt wird. Ist i kleiner oder gleich n (if_icmple: integer Vergleich, less or equal) wird der PC auf 7 gesetzt und mit der Ausführung des zugehörigen iload_2 Befehls begonnen. Dieser legt die Variable res auf den Operandenstapel. Der anschließende iload_3 Befehl schreibt die lokale Variable 3, also i auf den Operandenstapel. Diese beiden Werte werden von dem iadd Befehl vom Stapel genommen, addiert und das Ergebnis wieder auf den Operandenstapel gelegt. Anschließend wird dieses Ergebnis wieder in die lokale Variable 2 (res) geschrieben.

Damit ist die Java Quelltextzeile res += i abgearbeitet. Am Ende einer jeden for Schleife muß die Schleifenvariable noch verändert werden. Dies übernimmt der iinc Befehl, der in unserem Beispiel die lokale Variable 3 um 1 erhöht.

Zu guter letzt wird wieder die Schleifenbedingung überprüft, und je nach Ergebnis an die entsprechende Stelle gesprungen, bzw. mit der Ausführung des nachfolgenden Befehls fortgefahren. Dieser nachfolgende Befehl lädt die lokale Variable 2 (res) und liefert den Wert als Ergebnis dieses Methodenaufrufs.

Für die direkte Programmierung der JVM-Opcodes existieren Java-Assembler wie *Jasmin*. Die Syntax lehnt sich an die [aus Kapitel 1 bekannte](#) Ausgabe des Java-Disassemblers javap an.

Mit Java steht jedoch keineswegs die einzige Hochsprache zur Erzeugung von Byte-Code-Dateien zur Verfügung. [Diese Seite](#) listet eine Vielzahl verschiedener Alternativen.

Beispielsweise erzeugt der [Oberon-Compiler von Canterbury](#) für alle Oberon-Module, einschließlich der Systemmodule, Java-Klassen.

```

MODULE helloworld;

IMPORT Out;

BEGIN
    Out.String( "Hello World" );
    Out.Ln;
END helloworld.

```

Beispiel 57: Die Hello World Applikation als Oberon Programm [helloworld.obn](#)

Die erzeugten class-Dateien -- [SYSTEM.class](#), [helloworld.class](#), [Out.class](#), [Sys.class](#) -- können auf jeder JVM zur Ausführung gebracht werden. `java helloworld` liefert das erwartete Ergebnis.

Das **Class-File-Format**

Ausgangspunkt jeder Programmausführung innerhalb der JVM ist die class-Datei als Eingabe. Sie wird üblicherweise durch den Java-Compiler (im JDK: [javac](#) erzeugt).

Einige Eigenschaften jedes class-Files:

- Es enthält die Definition genau einer Klasse oder einer Schnittstelle, unabhängig von deren Zugriffs- und Sichtbarkeitseigenschaften.
- Es wird als Bytestrom aufgefaßt, daher werden alle größeren Informationseinheiten durch serielles Lesen und aneinanderfügen gebildet.
Unabhängig von der physischen Realisierung der tatsächlich zugrundeliegenden Hardware werden alle Multibyte-Daten im *big endian*-Format, d.h. größere Einheiten werden durch Linksshift und Boole'sches ODER gebildet, abgelegt.
Die Schnittstellen [java.io.DataInput](#), [java.io.DataOutput](#), [java.io.DataInputStream](#) und [java.io.DataOutputStream](#) unterstützen dieses Format. ([Beispiel](#))
- Die Strukturen innerhalb der class-Datei werden nicht zusätzlich optimiert abgelegt, daher erfolgt weder ein Auffüllen auf spezifische Wortgrenzen, noch ein Alignment an solchen.

Die JVM-Spezifikation legt zur Definition der Struktur des class-Files [eigene Datentypen](#) fest: `u1`, `u2` und `u4` zur Definition vorzeichenloser ein-, zwei- und drei-Bytetypen. Für diese (von der Java-üblichen vorzeichenbehafteten Mimik (abgesehen von `char`) abweichenden) Datentypen stehen mit [readUnsignedByte\(\)](#), [readUnsignedShort\(\)](#) und [readInt\(\)](#) entsprechende Lesemethoden zur Verfügung.

[The class File Format @ Java Virtual Machine Specification](#)

```

ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}

```

[Constant Pool @ Java Virtual Machine Specification](#)

```

cp_info {
    u1 tag;
    u1 info[];
}

```


field_info @ Java Virtual Machine Specification

```

field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}

```

method_info @ Java Virtual Machine Specification

```

method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}

```

attribute_info @ Java Virtual Machine Specification

```

attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}

```

Aufbau einer class-Datei verdeutlicht an nachfolgendem Beispiel Quellcode.

Hinweis: Mit [Classeditor](#) existiert ein freies Werkzeug zur Inspektion und Modifikation übersetzter Class-Dateien.

```

(1) class Act {
(2)     public static void doMathForever() {
(3)         int i=0;
(4)         while (true) {
(5)             i += 1;
(6)             i *= 2;
(7)         } //while
(8)     } //doMathForever()
(9)} //class Act

```

Beispiel 58: Java-Quellcode der untersuchten Klassendatei [Snipet.java](#)

magic



Abbildung 16: magic Identifier (click on image to enlarge!)

Der magic-Identifier ist auf die Bytekombination (in hexadezimaler Darstellung) CA FE BA BE fixiert. Anhand dieser erkennt der Kassenlader der Laufzeitsystems die Datei als ausführbare Java-Bytecode-Datei an.

Ist diese gegenüber dem Vorgabewert modifiziert wird eine `java.lang.ClassFormatError` Ausnahme im Hauptthread generiert (Bad magic number wird als zusätzliche Nachricht der Ausnahme ausgegeben). [siehe JVM-Spezifikation](#)

version identifier
minor major



Abbildung 17: version identifier (click on image to enlarge!)

Die beiden Versionskennungen minor und major bilden gemeinsam den Versionsschlüssel der class-Datei in der gängigen Punktnotation. Hierbei gilt: major.minor

Die Klassendatei des Beispiels trägt den Versionsschlüssel 45.3.

Der im SUN JDK enthaltene Java-Compiler erlaubt per Kommandozeilenparameter (target) die JVM-spezifische Steuerung der Codegenerierung. Die Auswahl von 1.1 als Zielplattform generiert die Versionskennung 45.3, 1.2 -- 46.0, 1.3 -- 47.0 und 1.4 (mit den Abarten 1.4.1 und 1.4.2) sowie der erste Prototyp des 1.5-Compilers -- 48.0. Die zweite Vorversion generierte bereits Bytecode mit der Versionsnummer 50.0, während die erste regulär verfügbare Betaversion die Versionsnummer 49.0 erzeugt.

Vorgabegemäß wird durch die Compilerversion 1.4.2 das Format für 1.2 und ab Version 1.5 48.0 erzeugt.

Trägt eine class-Datei eine durch die JVM nicht unterstützte Versionsnummer, so wird eine java.lang.UnsupportedClassVersionError-Ausnahme generiert.

Jede JVM kann verschiedene class-Datei-Versionen unterstützen, die letztendlich Festlegung welche Versionen durch einzelne Java-Plattform-Releases zu unterstützten sind obliegt jedoch SUN. So unterstützt SUNs JDK v1.0.2 class-Dateien der Versionen 45.0 bis einschließlich 45.3. Die JDK-Generation v1.1.x ab Version 45.0 bis einschließlich 45.65535 und Implementierungen der Java 2 Plattform, Version 1.2, bis einschließlich 46.0. JDK v1.3.0 verarbeitet Klassendateien bis hin zur Versionsnummer 47.0. Zur Verarbeitung von Klassen, welche die in 1.5 eingeführten Generizitätsmechanismen verwenden nicht zwingend eine Ausführungsumgebung dieser Versionsnummer benötigt, da das erzeugte Klassenformat (bisher, da diese Aussagen auf dem Informationsstand der verfügbaren Betaversion basieren) nicht verändert wurde. Die Nutzung des dynamischen Boxing/Unboxings benötigt jedoch eine Ausführungsumgebung mindestens der Version 1.5.

[siehe JVM-Spezifikation](#)

constant pool count

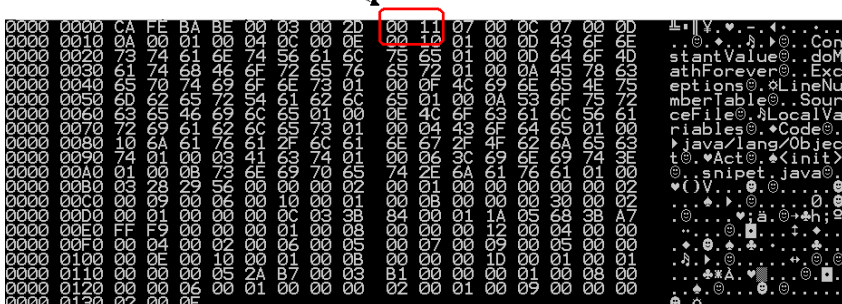


Abbildung 18: constant pool count (click on image to enlarge!)

Die Bytefolge constant_pool_count enthält die um eins erhöhte Anzahl der Einträge der constant_pool Tabelle.

Im Beispiel ist dies: 17.

[siehe JVM-Spezifikation](#)

Der constant pool enthält die symbolische Information über Klassen, Schnittstellen, Objekte und Arrays. Die Elemente dieser Datenstruktur sind vom Typ cp_info und variieren je nach tag in ihrer Länge. Als Konstantentypen (= Inhalt des Tag-Bytes) sind zugelassen:

Konstantentyp**Wert**

CONSTANT_Utf8	1
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_NameAndType	12
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_CLASS	7
CONSTANT_String	8
CONSTANT_Fieldref	9

[siehe JVM-Spezifikation](#)

1. Element des constant_pool (Superklassen-Verweis)

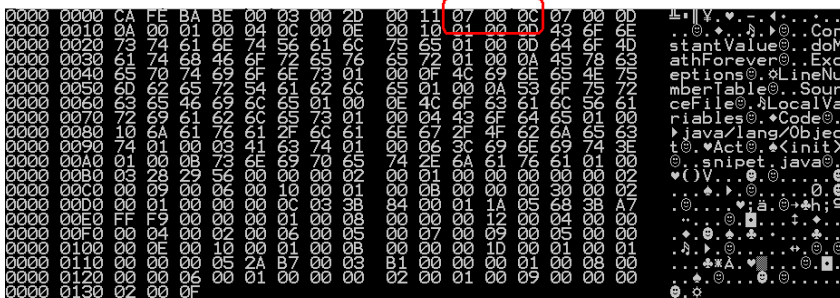


Abbildung 19: erstes Element des constant pools / Referenz auf Superklasse (click on image to enlarge!)

Konstante vom Typ `CONSTANT_class` die einen Verweis auf auf das zwölfte Element des Konstantenpools (0x0C) enthält. Zum Lesezeitpunkt der ersten Konstante kann diese Referenz noch nicht aufgelöst und auf Gültigkeit geprüft werden. (Später sehen wir, daß es sich um eine Referenz auf `java.lang.Object` handelt).
Hierbei handelt es sich immer um die Referenz auf die Superklasse. Auch für die API-Klasse `Object` selbst findet sich diese Refenz in der Klassendatei, auch wenn Diassemblierungswerkzeuge wie `javap` diese nicht ausgeben.

2. Element des constant_pool

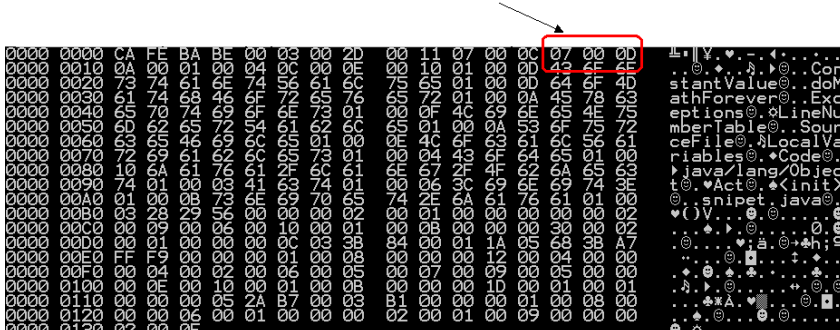


Abbildung 20: zweites Element des constant pools / this Referenz (click on image to enlarge!)

Konstante vom Typ `CONSTANT_class` die einen Verweis auf auf das 13. Element des Konstantenpools (0x0D) enthält. An dieser Stelle findet sich der String `Act`, also der Klassenname der zur Klassendatei gehörigen Klasse selbst. Auch hierbei handelt es sich zunächst um eine nicht auflösbare Vorwärtsreferenz.

Technisch gesehen realisiert sie den `this`-Verweis.

3. Element des constant_pool (Konstruktorenreferenz)

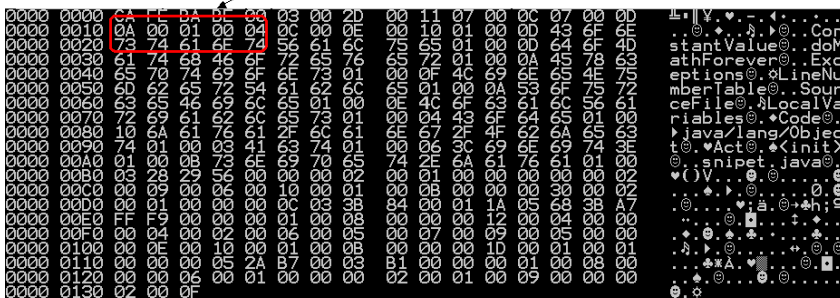


Abbildung 21: drittes Element des constant pools / Konstruktoren-Verweis (click on image to enlarge!)

Konstante vom Typ `CONSTANT_Methodref`. Die folgenden zwei Bytes (im Beispiel: 0x00 01) bezeichnen das referenzierte Objekt, gefolgt vom Methodenindex (0x00 04). Im Beispiel handelt es sich um das Objekt mit der Referenznummer 1 (=erstes Element des Konstantenpools, die Superklasse `Object`). Unter der Referenznummer 0x04 wird auf die Methode `<init>()` verwiesen. Da die betrachtete Klasse `Act` keinen eigenen Konstruktor definiert, wird der der Superklasse aufgerufen.

4. Element des constant_pool (Rückgabetypp des 14. Elements)

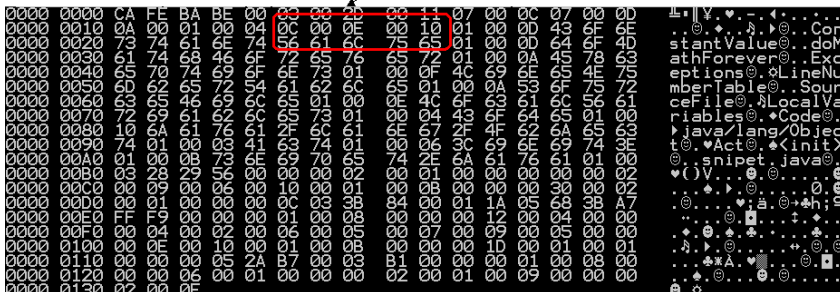


Abbildung 22: viertes Element des constant pools / Rückgabetypp des 14. Elements (click on image to enlarge!)

Konstante vom Typ `CONSTANT_NameAndType`. Die folgenden beiden Bytes (0x00 0E) verweisen auf die zu beschreibende Position innerhalb des Konstantenpools (im Beispiel: `init`). Dieser Position wird der and Position 0x10 spezifizierte Typ als Rückgabetypp zugeordnet -- im Beispiel `()V`; also `void`.

5. Element des constant_pool (konstante Zeichenkette)

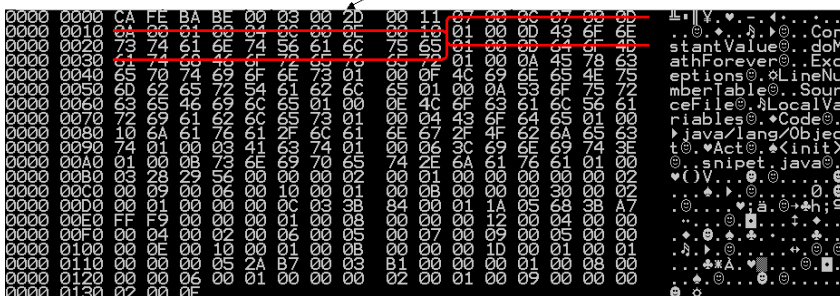


Abbildung 23: fünftes Element des constant pools / Konstante Zeichenkette (click on image to enlarge!)

Konstante vom Typ `CONSTANT_Utf8` leitet einen konstanten Zeichenketten-Ausdruck ein. Zeichenketten werden generell im Unicode UTF-8 Format abgelegt, wobei jedoch aus Speicherplatzeffizienzgründen für die Zeichen zwischen `\u0001` und `\u007F` nur jeweils ein Byte benötigt werden. Für alle anderen Unicode-Symbole wird der entsprechende 2- bzw. 3-Byte Speicherplatz zur Verfügung gestellt. Die Konstante wird von der Länge der Zeichenkette (im Beispiel: 13) gefolgt, daher kann auf eine terminierende Null verzichtet werden. Die Bedeutung dieser -- im Java-Quellcode nicht enthaltenen -- Zeichenkette wird im Kontext des Klassenladevorgangs deutlich.

10. Element des constant_pool (Zeichenkette "LocalVariable")



Abbildung 28: zehntes Element des constant pools / Zeichenkette LocalVariables (click on image to enlarge!)

Konstante vom Typ CONSTANT_Utf8, die den fixen String *LocalVariables* einleitet.

11. Element des constant_pool (Zeichenkette "Code")

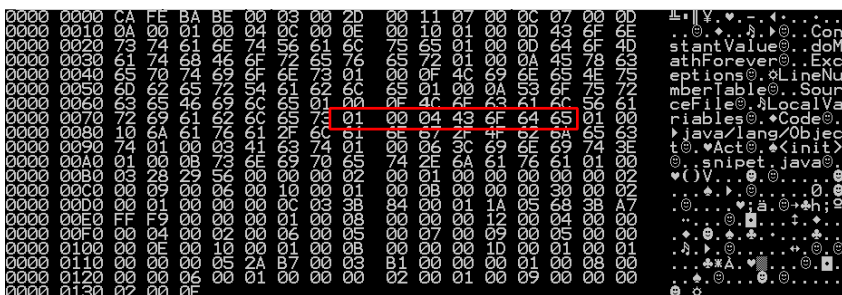


Abbildung 29: elftes Element des constant pools / Zeichenkette Code (click on image to enlarge!)

Konstante vom Typ CONSTANT_Utf8, die den fixen String *Code* einleitet.

12. Element des constant_pool (Zeichenkette "java/lang/Object")

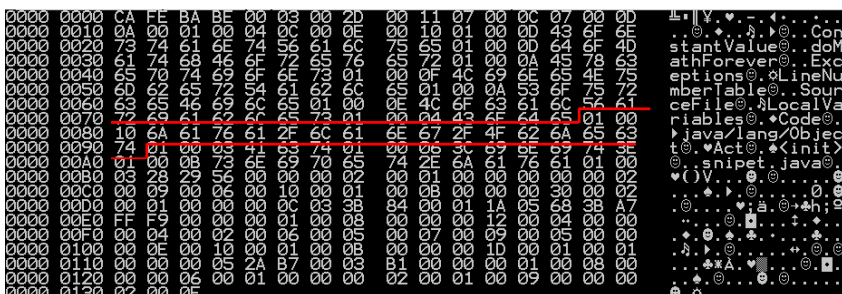


Abbildung 30: zwölftes Element des constant pools / Zeichenkette java.lang.Object (click on image to enlarge!)

Konstante vom Typ CONSTANT_Utf8, die den String *java/lang/Object* einleitet. Vom ersten Element des Konstantenpools referenziertes Element. Die in der Java-Hochsprache üblichen Punkte zur Trennung der Pakete, Subpakete und Klassennamen werden in der JVM konsequent (aus historischen Gründen) durch Querstriche ersetzt.

13. Element des constant_pool (Zeichenkette "Act")

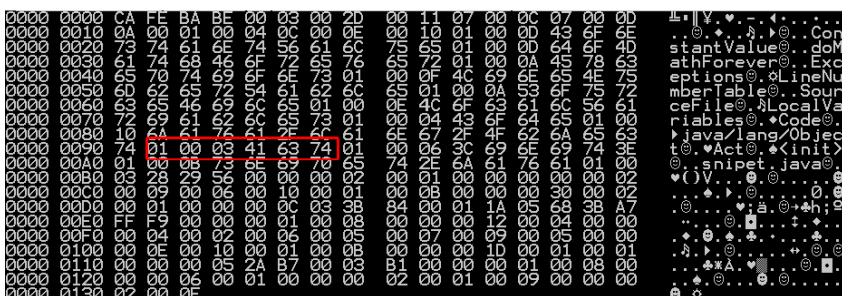


Abbildung 31: 13. Element des constant pools / Zeichenkette Act (click on image to enlarge!)

Konstante vom Typ `CONSTANT_Utf8`, die den String `Act` einleitet. Vom zweiten Element des Konstantenpools referenziertes Element. Name der Klasse.

14. Element des constant_pool (Zeichenkette "<init>")



Abbildung 32: 14. Element des constant pools / Zeichenkette `init` (click on image to enlarge!)

Konstante vom Typ `CONSTANT_Utf8`, die den String `<init>` einleitet. Methodenname der innerhalb der Superklasse `Object`. Der Rückgabewert dieser Methode ist im vierten Element des Konstantenpools abgelegt.

15. Element des constant_pool (Zeichenkette "snipet.java")

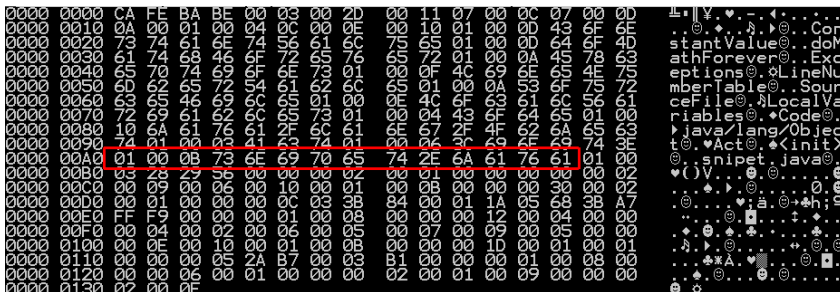


Abbildung 33: 15. Element des constant pools / Zeichenkette `snipet.java` (click on image to enlarge!)

Konstante vom Typ `CONSTANT_Utf8`, die den String `snipet.java` -- den Namen der Quellcodedatei in der sich die Definition der Klasse `Act` befindet -- einleitet.

16. Element des constant_pool (Zeichenkette "()V")

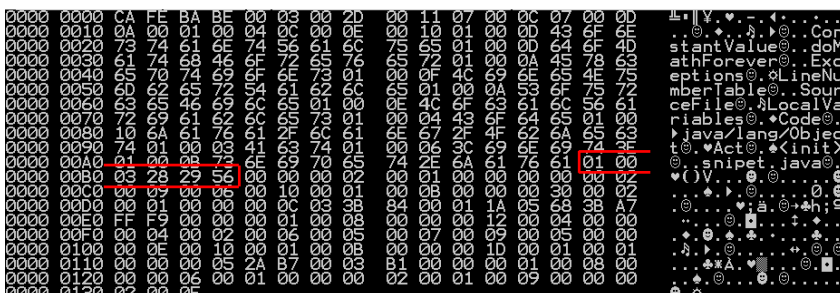


Abbildung 34: 16. Element des constant pools / Zeichenkette `()V` (click on image to enlarge!)

Konstante vom Typ `CONSTANT_Utf8`, die den String `()V` einleitet. Methodendeskriptor, der weder Übergabeargumente noch Rückgabebetyp besitzt.

Zugriffsflags

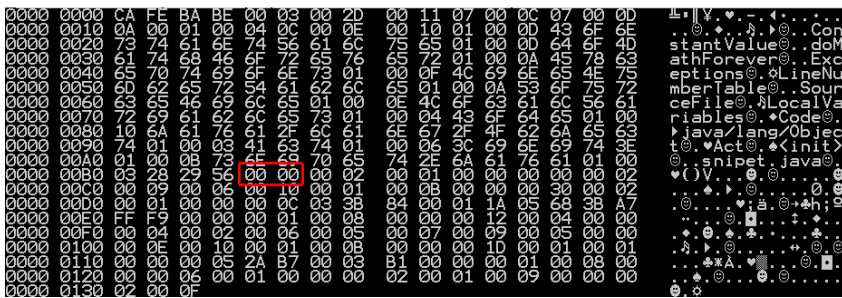


Abbildung 35: Zugriffsflags (click on image to enlarge!)

Zugriffsflags für die Klasse Act. Der konkrete Code ergibt sich aus der binären ODER-Verknüpfung verschiedener Zugriffsflaggen, die in untenstehender Tabelle wiedergegeben sind.

Flag	Wert	Beschreibung
ACC_PUBLIC	0x0001	public-Deklaration; Zugriffbar von allen anderen Klassen, auch außerhalb des eigenen Pakets
ACC_FINAL	0x0010	final-Deklaration; Verbot der Vererbung
ACC_SUPER	0x0020	Besondere Behandlung der Superklasseninstruktionen bei Aufruf über Opcode invokespecial
ACC_INTERFACE	0x0200	Struktur ist Schnittstelle, keine Klasse
ACC_ABSTRACT	0x0400	Abstrakte Struktur, von der keine Ausprägungen erzeugt werden können

Referenz auf this und super

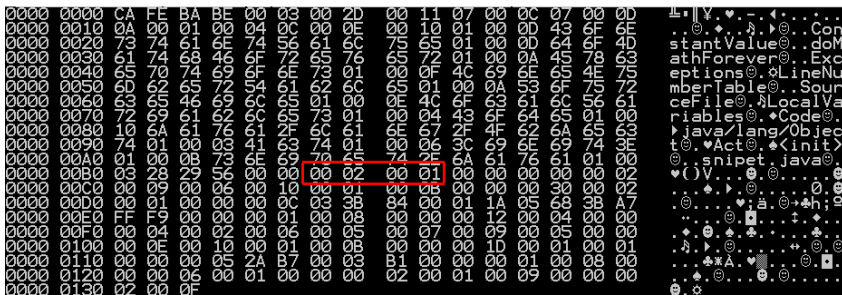


Abbildung 36: Referenz auf this und super (click on image to enlarge!)

Referenz in den Konstantenpool, auf die Klasse selbst (this) und die Superklasse (super).

interface_count und
fields_count

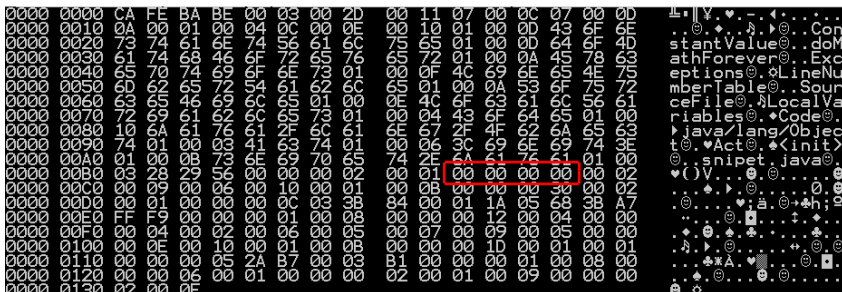


Abbildung 37: interface_count und fields_count (click on image to enlarge!)

interface_count: Anzahl der durch die Klasse implementierten Schnittstellen.

fields_count: Anzahl der Klassen- oder Instanzvariablen der Klasse.

methods_count

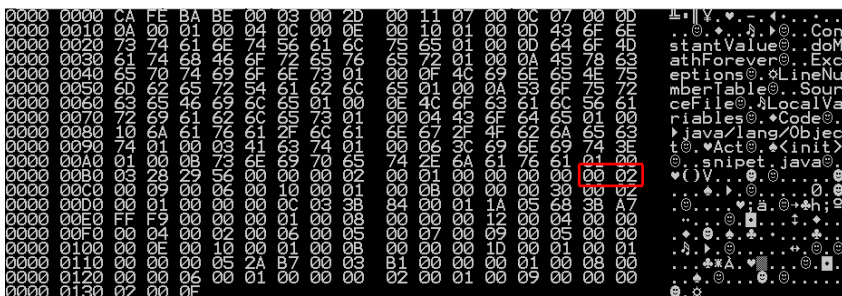


Abbildung 38: methods_count (click on image to enlarge!)

Anzahl der durch die Klasse implementierten Methoden.

Die Zahl ergibt sich aus der tatsächlich durch den Anwender definierten Anzahl und den impliziten, d.h. durch den Compiler hinzugefügten (z.B. Konstruktor), Methoden.

methods_info (doMathForever())

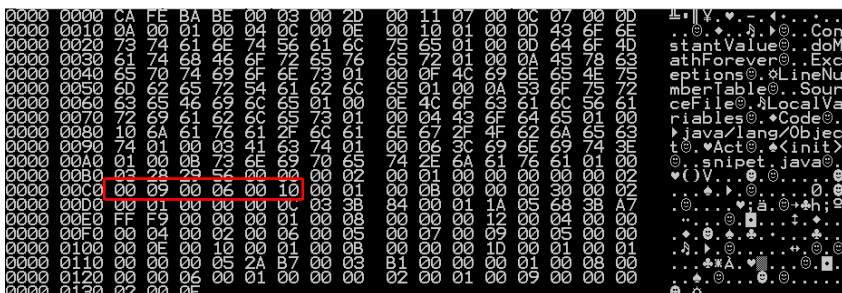


Abbildung 39: methods_info (click on image to enlarge!)

Informationen über die in der Klasse Act definierten Methoden.

Die Zugriffsflaggen (0x00 09) weisen doMathForever() als static und public aus. (Die konkrete Wertebelegung kann untenstehender Aufstellung entnommen werden. Auch in diesem Falle wird der tatsächliche Wert durch Boole'sche ODER-Verknüpfung der Einzelwerte gebildet.)

Der Verweis (0x06) auf das sechste Element des Konstantenpools referenziert den Methodennamen im Klartext. Der zweite Verweis (0x10) auf den Konstantenpool kennzeichnet doMathForever() als parameterlose Methode ohne Rückgabetyphen.

Flag	Wert	Beschreibung
ACC_PUBLIC	0x0001	public-Deklaration; Zugriffbar von allen anderen Klassen, auch außerhalb des eigenen Pakets
ACC_PRIVATE	0x0002	Als private deklariert, daher nur innerhalb der definierenden Klasse verwendbar
ACC_PROTECTED	0x0004	protected-Deklaration, Zugriff nur in Subklassen möglich
ACC_STATIC	0x0008	static-Deklaration, keine Auswirkungen auf Sichtbarkeit und Zugriffsrechte
ACC_FINAL	0x0010	final-Deklaration; nach initialer Zuweisung keine Wertänderung möglich
ACC_VOLATILE	0x0040	volatile-Deklaration; keine Berücksichtigung in Optimierungsmaßnahmen
ACC_TRANSIENT	0x0080	transient-Deklaration; keine Berücksichtigung durch Persistenzmanager

Attribute der Methode doMathForever()

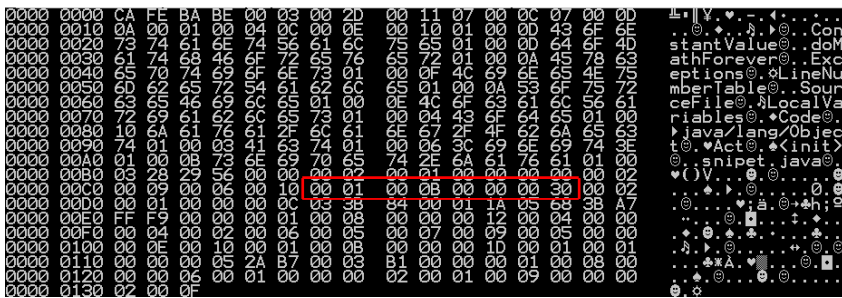


Abbildung 40: Attribute der Methode doMathForever() (click on image to enlarge!)

Die Methode `doMathForever()` verfügt nur über genau ein Attribut, daher ist der attribute `count` zu Beginn der Byte-Sequenz auf `0x00 01` gesetzt. Dieses eine Attribut wird durch Index 11 innerhalb des Konstantenpools referenziert. Dort ist die Zeichenkette `Code` lokalisiert. Dadurch wird angezeigt, daß die folgenden Bytes die Implementierung dieser Methode beinhalten. Der abschließende vier-Byte Indikator enthält die Länge der Methodenimplementierung (im Beispiel: `0x30`).

maxStack und maxLocals der Methode doMathForever()

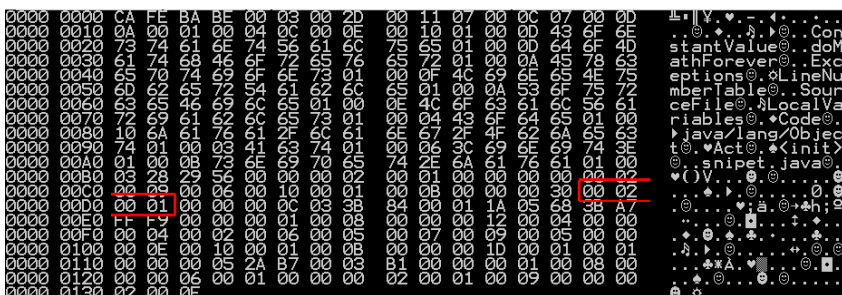


Abbildung 41: maxStack und maxLocals der Methode doMathForever() (click on image to enlarge!)

maxStack: Maximalhöhe des Operandenstacks die während der Methodenausführung erreicht werden kann. (Im Beispiel: 2; die Opcode-Implementierung der beiden verwendeten arithmetischen Operationen benötigen niemals mehr als zwei Stackpositionen.)
maxLocals: Anzahl der lokalen Variablen. (die verwendete Variable `i`)

Bytecode und Ausnahmentabelle der Methode doMathForever()

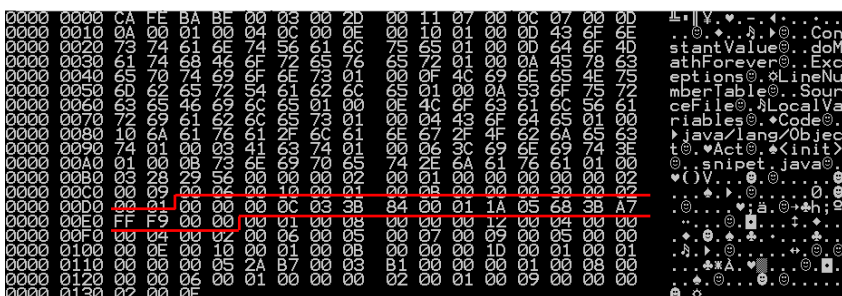
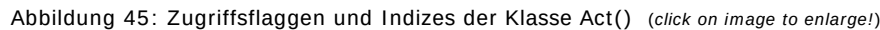


Abbildung 42: Bytecode und Ausnahmentabelle der Methode doMathForever() (click on image to enlarge!)

Die `code length` legt die Anzahl der folgenden Bytecode-Instruktionen fest (im Beispiel: 12), darauf folgen die tatsächlichen Opcodes.

pc	instruction	mnemonic
0	03	iconst_0
1	3B	istore_0
2	840001	iinc 0 1
5	1A	iload_0
6	05	iconst_2
7	68	imul
8	3B	istore_0
9	A7FFF9	goto 2

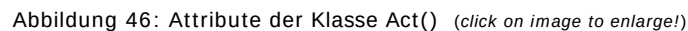
Act()



Die Referenz in den Konstantenpool verweist auf die implementierende Methode (im Beispiel: Position 0x0E, dort findet sich die Methode <init>).

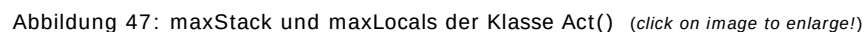
Attribute der Klasse

Act()



Der abschließend angegebene Längenzähler fixiert die Anzahl der folgenden Bytes.

Act()



<http://www.jeckle.de/vorlesung/java/script.html> (87 of 162)09.01.2004 07:43:46

Bytecode und Ausnahmetabelle der Klasse

Act()

pc	instruction	mnemonic
0	2A	aload_0
1	B70003	invokeonvirtual #3 <Method java.lang.Object <init> ()V>
4	B1	return

Abbildung 48: Bytecode und Ausnahmetabelle der Klasse Act() (click on image to enlarge!)

Opcode-Implementierung des Konstruktors, sowie die Aufzählung der durch ihn potentiell ausgelösten Ausnahmeereignisse (im keine, daher Anzahl gleich Null).

Die Implementierung in Java-Bytecode:

pc	instruction	mnemonic
0	2A	aload_0
1	B70003	invokeonvirtual #3 <Method java.lang.Object <init> ()V>
4	B1	return

Eigenschaften des Code-Bereichs der Klasse

Act()

pc	instruction	mnemonic
0	2A	aload_0
1	B70003	invokeonvirtual #3 <Method java.lang.Object <init> ()V>
4	B1	return

Abbildung 49: Eigenschaften des Code-Bereichs der Klasse Act() (click on image to enlarge!)

Anzahl der Eigenschaften im ersten Doppelbyte (im Beispiel: 1). Die spezifische Eigenschaft wird durch Index acht im Konstantenpool (=LineNumberTable) näher definiert.

Diese Tabelle hat die Länge 0x06, mit einem einzigen Eintrag.

Zuordnung zwischen LineNumberTable und der Klasse

Act()

pc	instruction	mnemonic
0	2A	aload_0
1	B70003	invokeonvirtual #3 <Method java.lang.Object <init> ()V>
4	B1	return

Abbildung 50: Zuordnung zwischen LineNumberTable und der Klasse Act() (click on image to enlarge!)

Zuordnung der Quellcodezeilennummern zu den resultierenden Opcodes.

Allgemeine Eigenschaften

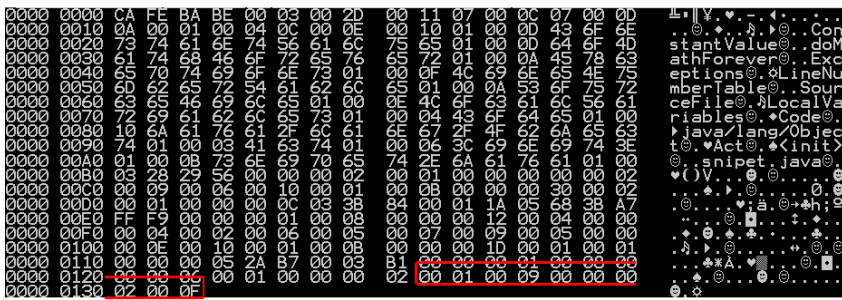


Abbildung 51: Allgemeine Eigenschaften (click on image to enlarge!)

Am Ende einer Klassendatei kann eine beliebige Menge allgemeiner Attribute angegeben werden. für das Beispiel wurde durch den Compiler genau ein Attribut erzeugt, ein Verweis auf die Quelldatei (Konstantenpool-Index 0x09). Auf diese Information folgt ein Verweis auf den Namen der Quellcodedatei (Konstantenpool-Index 0x15).

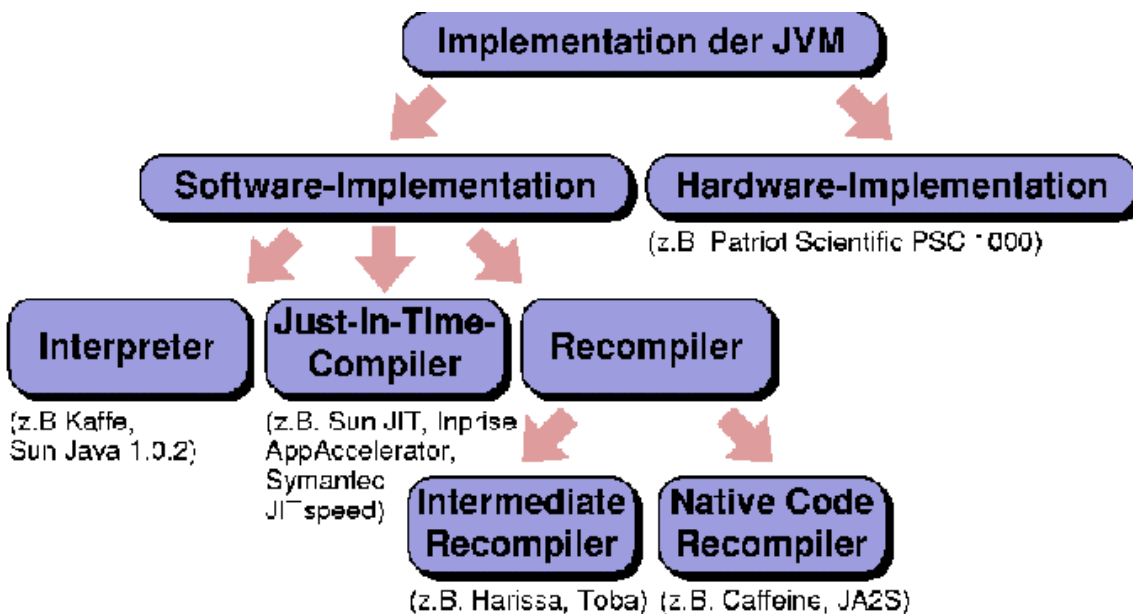
Schlußbemerkungen:

Abbildung 52: Implementationsansätze für die Execution Engine der JVM (click on image to enlarge!)

Graphik aus: [Raner, M.: Blick unter die Motorehaube](#)

Die interpretative Ausführung einer class-Datei mit der virtuellen Java-Maschine ist jedoch keineswegs zwingend, auch wenn sie das derzeit am häufigsten anzutreffende Vorgehen verkörpert. Bereits in der Standardedition des Java Development Toolkits von SUN wird seit Version 1.2 ein *just in time compiler* mitgeliefert, der transparent in die Ausführungsumgebung integriert ist. Er durchbricht die befehlweise interpretative Abarbeitung und greift den bei der Abarbeitung dynamisch durch den Interpretationsprozeß entstehenden plattformspezifischen Maschinencode ab und puffert ihn zwischen. Bei jeder erneuten Ausführung derselben Bytecodesequenz wird nun dieser bereits übersetzte Code ausgeführt. Dieses Vorgehen ist in den verbreiteten JVM-Implementierungen der Internet-Browser von Netscape und Microsoft verwirklicht. Ebenso bieten fast alle verfügbaren Java-Entwicklungsumgebungen diese Laufzeit-optimierende Komponente an. Der dadurch erzielte Geschwindigkeitsvorteil bewegt sich, je nach Struktur der Anwendung, zwischen zehn und 50 Prozent.

Den größten Geschwindigkeitszuwachs verspricht man sich jedoch von der vollständigen Realisierung der virtuellen Maschine in Hardware; damit wird sie de facto zur realen Maschine. Hierzu liegen jedoch noch keine Ergebnisse vor, welche die der derzeitigen Implementierung auf handelsüblichen Prozessoren signifikant überträfen.

Eine andere Sichtweise nutzt das Bytecodeformat welches eine Zwischenrepräsentation darstellt nicht zur Interpretation mit dem Ziele der direkten Ausführung, sondern als Eingabeformat eines weiteren Übersetzungsschrittes, der üblicherweise plattformabhängigen nativen Code erzeugt. Für C++ existieren bereits Umsetzungen, die Bytecode in übersetzungsfähigen C++-Quellcode transformieren. Eine Spielart hiervon bildet das diskutierte Werkzeug *javap* dessen Ausgabeformat, nach einigen Umformungen, direkt als Eingabe weiterer Übersetzer akzeptiert wird.

3.2 Die Java API und weiterführende Themen

3.2.1 Ein- / Ausgabe -- Streams

Zur Erinnerung: **bisher behandelte Möglichkeiten zur Ein- und Ausgabe:**

Bisher wurden ausschließlich Bildschirm-Ausgaben, und diese ausschließlich mit der statischen Methode `System.out.println`, erzeugt werden. Der einzige uns bisher bekannte Mechanismus zur Eingabebehandlung war der der Kommandozeilenparameter in der `main`-Methode.

Wie aus C++ bekannt verfügt auch Java über die objektorientierte Kapselung der Ein- und Ausgabebehandlung in Form von *Streams*. Hierbei stehen beliebigste Ein- und Ausgabequellen über denselben programmiersprachlichen Mechanismus zur Verfügung, unabhängig davon wie das physische Gerät realisiert ist.

Zentrales Paket für die Ein-/Ausgabebehandlung ist [java.io](#). Es enthält neben den wichtigsten Klassen zur Implementierung des E/A-Verhaltens auch verschiedene Schnittstellen, sowie die möglichen Ausnahmeereignisse während der verschiedenen E/A-Operationen. Gegenüber den aus C++ bekannten Datenströmen tritt bei Java hinzu, daß auch Informationen über Netze mit denselben Mechanismen übertragen werden.

In Java werden generelle zwei Streamtypen unterschieden: **Character Streams** und **Byte Streams**. Wie der Name schon andeutet, sind Byte Streams auf die Verarbeitung von beliebigen Byte-artigen Informationseinheiten -- und damit acht Bit große Einheiten -- beschränkt. Diese Mimik stellt insbesondere bei der Verarbeitung von Unicode-Zeichenketten eine große Einschränkung dar, da hierbei nicht gesamte Zeichen-Information (ein Symbol mißt 16 Bit) in einem Zugriff verarbeitet werden kann. Daher wurde mit dem JDK v1.1 zusätzlich das Konzept der Character Streams eingeführt, die generell 16 Bit lange Zeichen bereitstellen.

Als **Byte Streams** stehen zur Verfügung:

(Eindrückungen kennzeichnen Subklassenbeziehungen, Kursivsetzungen abstrakte Klassen)

- [InputStream](#)
Abstrakte Basisklasse der meisten Eingabeströme. Sie definiert die Signatur die [grundlegenden Eingabemethoden](#).
 - [FileInputStream](#)
Byteweises Lesen einer Datei.
 - [PipedInputStream](#)
Hochsprachliches Äquivalent der Betriebssystem Pipes. Einsatzfeld: Gemeinsam mit [PipedOutputStream](#) zum Datenaustausch zwischen verschiedenen [Threads](#) derselben Applikation.
 - [FilterInputStream](#)
Gemeinsam mit anderen Eingabeströmen eingesetzt. Einsatzgebiet: Nachbearbeitung roher Eingaben. Die entsprechenden Subklassen stellen konkrete Implementierungen häufiger Einsatzfälle dar.
 - [LineNumberInputStream](#)^{deprecated}
Liefert zur Eingabe die zugehörige Zeilennummer des Eingabestroms.
Achtung: Diese Klasse geht von der (i. A. falschen) Entsprechung zwischen Bytes und Charactern aus. [LineNumberReader](#) liefert für denselben Anwendungsfalle eine korrekte Implementierung.
 - [DataInputStream](#)
Liest [primitive Java-Datentypen](#) plattformunabhängig aus einem Datenstrom.
 - [BufferedInputStream](#)
Verleiht beliebigen [InputStreams](#) die Möglichkeit des gepufferten Lesens aus dem Eingabestrom. Zusätzlich wird das Setzen von, und die Rücksetzung des Strompositionszeigers zu, Markierungspunkten unterstützt.
 - [PushbackInputStream](#)
Verleiht beliebigen [InputStreams](#) die Möglichkeit der Wiedereinstellung von bereits gelesenen Byte-Inhalten in den Eingabestrom.
 - [ByteArrayInputStream](#)
Lesen aus einem Byte Array.
 - [StringBufferInputStream](#)^{deprecated} Lesen aus `String`. *Achtung:* Diese Klasse geht von der (i. A. falschen) Entsprechung zwischen Bytes und Charactern aus. [StringReader](#) liefert für denselben Anwendungsfalle eine korrekte Implementierung.
 - [SequenceInputStream](#)
Konkatenation von Eingabeströmen; nach vollständigem Lesen des *i*-ten Eingabestroms wird der *i*+1-te angesprochen.

- [ObjectInputStream](#)
Lesen vollständig serialisierter Objekte, die die Schnittstellen [java.io.Serializable](#) oder [java.io.Externalizable](#) unterstützen.
- [OutputStream](#)
Abstrakte Basisklasse der meisten Ausgabeströme. Sie definiert die Signatur der [grundlegenden Ausgabemethoden](#).
 - [FileOutputStream](#)
Schreibt Byte-artige Daten in Dateien oder durch Dateideskriptoren bezeichnete Ziele.
 - [PipedOutputStream](#)
Umkehrung (Schreibende) eines [PipedInputStreams](#).
 - [FilterOutputStream](#)
Verleiht Ausgabeströmen die Möglichkeit Ausgabedaten zu verändern. Einige konkrete Implementierungen sind durch die Subklassen gegeben.
 - [DataOutputStream](#)
Erlaubt plattformunabhängiges Schreiben der [primitiven Java-Datentypen](#).
 - [BufferedOutputStream](#)
Verleiht verschiedenen [OutputStreams](#) die Möglichkeit des gepufferten (d.h. nicht mehr Byte-weisen) Schreibens.
 - [PrintStream](#)
Verleiht anderen [OutputStreams](#) diverse Möglichkeiten zur Ausgabe der verschiedenen Java Datentypen. Anders als die übrigen Stromtypen erzeugt PrintStream keine [IOException](#).
Anmerkung: Es werden Bytes, keine Characters geschrieben. Mit [PrintWriter](#) existiert eine Standardimplementierung, die auf die Besonderheiten im Zusammenhang mit Unicode-Ausgaben Rücksicht nimmt.
 - [ByteArrayOutputStream](#)
Ausgabestrom, der in einen Byte-Array schreibt.
 - [ObjectOutputStream](#)
Schreibt [Primitivtypen](#) und Objekte. Ausschließlich Objekte, welche die [java.io.Serializable](#)-Schnittstelle implementieren können über diesen Mechanismus persistent geschrieben werden.

Als **Character Streams** stehen zur Verfügung:

(Eindrückungen kennzeichnen Subklassenbeziehungen, Kursivsetzungen abstrakte Klassen)

- [Reader](#)
[Abstrakte Basisklasse der verschiedenen Reader-Implementierung. Sie definiert die Signatur der grundlegenden Character-bezogenen Ausgabemethoden.](#)
 - [BufferedReader](#)
Liest Textinhalte gepuffert aus einem Character-Eingabestrom.
 - [LineNumberReader](#)
Liest einzelne Unicode-Zeichen, Zeichensequenzen und ganze Zeilen gepuffert, und stellt zusätzlich die Zeilennummer des aktuell verarbeiteten Inhaltes.
 - [CharArrayReader](#)
Einlesen von Unicode-Zeichen aus Character-Puffer.
 - [InputStreamReader](#)
 - [FileReader](#)
Eingabestrom für character-Dateien.
 - [FilterReader](#)
Abstrakte Basisklasse verschiedener filternder Eingabeströme. Konkrete Implementierungen werden durch die verschiedenen Subklassen zur Verfügung gestellt.
 - [PushbackReader](#)
Unicode-Character Eingabestrom, der die Rückstellung von bereits gelesenen Inhalten in den Strom erlaubt.
 - [PipedReader](#)
Unicode-Character Eingabestrom als Lese-Ende einer Pipe.
 - [StringReader](#)
Unicode-Character Eingabestrom, der aus einem [String](#) liest.
- [Writer](#)
Abstrakte Basisklasse verschiedener Implementierungen zur Ausgabe von Unicode-Zeichenströmen.
 - [BufferedWriter](#)
Gepuffertes Schreiben. Zeilenvorschub wird transparent plattformabhängig umgesetzt. (siehe [system properties](#)).
 - [CharArrayWriter](#)
 - [OutputStreamWriter](#)
Brücke zwischen Character und Byte-Strömen; Unicode-Zeichen werden automatisch während des Schreibens in die entsprechende Byte-Repräsentation umgesetzt.
 - [FileWriter](#)
Schreibt Character-Dateien.
 - [FilterWriter](#)

Abstrakte Basisklasse der verschiedenen Implementierungen zum gefilterten Schreiben.

- [PipedWriter](#)
Ausgabestrom über Character-Pipe.
- [StringWriter](#)
Ausgabestrom, der in einen [String](#) schreibt.
- [PrintWriter](#)
Ausgabestrom der formatierte Objektrepräsentationen schreibt. Methoden dieser Klasse verursachen keine E/A-Ausnahmeereignisse.

Üblicherweise existieren die Ströme immer symmetrisch, d.h. in gleicher Weise sowohl für Ein- als auch Ausgabe. Dieses Prinzip wird nur für einzelne Klassen durchbrochen, die zwar Eingabeseitig existieren (beispielsweise Lesen mit Zeilennummer), für die jedoch kein expliziter Ausgabemechanismus benötigt wird.

Zusätzlich zu den nach Zugriffsarten klassifizierten Strömen existiert mit der Klasse [RandomAccessFile](#) eine Strom über den sowohl lesende als auch schreibende Zugriffe abgewickelt werden können.

Die abstrakten Basisklassen [InputStream](#) bzw. [Reader](#) und definieren ähnliche Lese- und Zugriffsmethoden, die in allen abgeleiteten Klassen auf den entsprechenden Stromtypen zur Verfügung stehen.

Die aktuell gewünschte Zugriffsart (nur-lesend, nur-schreibend oder beides) wird über einen Parameter des Konstruktors gesteuert.

Wie durch den Klassennamen bereits angedeutet, existiert dieser Strom ausschließlich für Dateien; eine Netzwerkanwendung ist nicht möglich.

grundlegende Lesemethoden:

- [java.io.InputStream](#):
 - [int available\(\)](#) Liefert die Anzahl Bytes die an der Eingabeschnittstelle zur Verfügung stehen. Diese Anzahl kann ohne Blockierung des Aufrufers gelesen werden.
 - [void close\(\)](#) Schließt Eingabestrom unter Freigabe der belegten Systemressourcen
 - [void mark\(int\)](#) Markiert die gegenwärtige Position des Eingabezeigers im Eingabestrom; ein folgender Aufruf von [reset](#) setzt den Eingabezeiger wieder an die markierte Position zurück.
 - [boolean markSupported\(\)](#) Gibt Auskunft darüber, ob der Eingabestrom die Positonsmarkierung, und das Rücksetzen darauf, unterstützt.
 - [int read\(\)](#) Liefert den nächsten Bytewert (>0 und <256) aus dem Eingabestrom. Ist das Ende des Eingabestromes erreicht, wird -1 retourniert (kein Ausnahmeereignis! Die Verwendung des [StreamTokenizers](#) ermöglicht hier ein handlicheres Vorgehen).
Subklassen von [InputStream](#) sind gezwungen diese Methode zu überschreiben.
 - [int read\(byte\[\]\)](#) Liefert eine Bytesequenz aus dem Eingabestrom, und legt sie im übergebenen Array ab. Die Anzahl der gelesenen Zeichen, bzw. -1 beim Erreichen den Eingabeendes, wird zurückgegeben.
 - [int read\(byte\[\] b, int off, int len\)](#) Liefert Bytesequenz der Länge len und speichert sie ab Position off im übergebenen Array.
 - [void reset\(\)](#) Setzt den Stromzeiger auf die Position der letzten vorhergehenden Markierung zurück, falls eine solche existiert.
 - [long skip\(long n\)](#) Versucht den Stromzeiger um n Bytepositionen vorzurücken. Die Anzahl der tatsächlich übersprungenen Bytes wird zurückgegeben.
- [java.io.Reader](#):
 - [void close\(\)](#) Schließt den Ausgabestrom.
 - [void mark\(int readAheadLimit\)](#) Markiert gegenwärtige Position des Eingabezeigers. Ein nachfolgender Aufruf von [reset](#) versucht den Positionszeiger auf die Markierte Stelle zurückzusetzen, falls die aktuelle Position nicht weiter als readAheadLimit Zeichen entfernt liegt.
 - [void markSupported\(\)](#) Gibt Auskunft darüber, ob der Eingabestrom die Positonsmarkierung, und das Rücksetzen darauf, unterstützt.
 - [int read\(\)](#) zur Extraktion genau eines Zeichens (16 Bit Character).
 - [int read\(char\[\] cbuf\)](#) zur Extraktion einer Sequenz, beginnend ab der aktuellen Stromzeigerposition.
 - [int read\(char\[\] cbuf, int off, int len\)](#) zur Extraktion einer Sequenz der Länge len oder weniger von Zeichen, und Ablage in Array beginnend ab der Position off.
 - [boolean ready\(\)](#) Liefert dieser Aufruf true zurück, so liegen weitere Eingaben vor, die durch ein folgendes read gelesen werden können.
 - [reset\(\)](#) Versucht den Positionszeiger auf die durch die letzte gesetzte Markierung bezeichnete Stelle zurückzurücken.
 - [long skip\(long n\)](#) Versucht n Zeichenpositionen zu überspringen. Die Anzahl der tatsächlich übersprungenen Positionen wird zurückgeliefert.

grundlegende Schreibmethoden:

- [java.io.OutputStream](#):
 - [void close\(\)](#) Schließt Ausgabestrom und gibt durch ihn belegte Systemressourcen frei.
 - [void flush\(\)](#) Leert Ausgabestrom und schreibt alle Pufferbereiche.
 - [void write\(byte\[\]\)](#) Schreibt Byte-Array in Ausgabestrom.
 - [void write\(byte\[\] b, int off, int len\)](#) Schreibt len Bytes eines Byte-Arrays ab Position off.
 - [void write\(int\)](#) Schreibt Bytewert in Ausgabestrom.
 Subklassen von OutputStream müssen diese dieser Methode überschreiben.
- [java.io.Writer](#):
 - [void close\(\)](#) Schließt Ausgabestrom nach erfolgter Pufferleerung (flushing).
 - Subklassen von Writer müssen diese Methode überschreiben.
 - [void flush\(\)](#) Leert Pufferbereiche.
 - [write\(int\)](#) Schreibt ein Unicode-Zeichen.
 - [write\(char\[\]\)](#) Schreibt Character-Array.
 - [write\(char\[\] cbuf, int off, int len\)](#) Schreibt Teil der Länge len eines Character-Arrays beginnend ab Position off.
 - [write\(String\)](#) Schreibt [String](#).
 - [write\(String str, int off, int len\)](#) Schreibt Teil der Länge len eines [Strings](#) beginnend ab Position off.

Mit den [Dateideskriptoren](#) [in](#), [out](#) und [err](#) stehen die aus C/C++ bekannten drei **Standardströme** zur Verfügung.

Diese standardmäßig geöffneten Ströme stehen während der Ausführungszeit jeder Applikation zur Verfügung.

```
(1)import java.io.FileDescriptor;
(2)import java.io.FileWriter;
(3)import java.io.IOException;
(4)
(5)public class PrintLn {
(6)    public static void main(String[] args) {
(7)        FileWriter fw = null;
(8)        try {
(9)            fw = new FileWriter(FileDescriptor.out);
(10)           for (int i=0; i < args.length; i++)
(11)               fw.write (args[i]+" ");
(12)           } catch (IOException ioe) {
(13)               System.out.println("cannot open stdout!");
(14)           } finally {
(15)               try {
(16)                   fw.close();
(17)               } catch (Exception e) {
(18)                   //ignore it
(19)               } //catch
(20)           } //finally
(21)       } //main()
(22)} //class PrintLn
```

Beispiel 59: Ausgabe auf standard out [PrintLn.java](#)

Das Programm öffnet erzeugt ein `FileWriter`-Objekt mit dem vorgegebenen Dateideskriptor `out`. Anschließend werden die Kommandozeilenparameter (allesamt Typ `String`) mit `write` ausgegeben. Alle Methoden der Klasse `FileWriter` können ein Ausnahmeereignis vom Typ [IOException](#) erzeugen. Daher müssen Operationen auf dem erzeugten Ausgabestrom durch `try`-Blöcke abgesichert werden.

Wird als Ausgabekanal des `FileWriter`-Stroms auf eine physikalische Datei ausgerichtet, so muß lediglich die Konstruktoranweisung zu `fw = new FileWriter("myFile.asc")` modifiziert werden. Alle E/A-Klassen interpretieren die Pfadangabe relativ zum aktuellen Verzeichnis. Die Verzeichnisseparatoren variieren plattformabhängig. Verzeichnistrenner und aktueller Katalog können über die [system properties](#) ermittelt werden.

Streamerzeugung und mögliche Datenquellen aller (nicht als deprecated gekennzeichneten) Streamtypen:

Strom	Erzeugbar aus
BufferedInputStream	InputStream
BufferedOutputStream	OutputStream
BufferedReader	Reader
BufferedWriter	Writer
ByteArrayInputStream	Byte Array
CharArrayReader	Character Array
DataInputStream	InputStream Beispiel
DataOutputStream	OutputStream
FileInputStream	File -Objekt FileDescriptor (nur auf die Standardströme stdin, stdout, stderr anwendbar) String , der einen gültigen Pfad enthält
FileOutputStream	File -Objekt, FileDescriptor (nur auf die Standardströme stdin, stdout, stderr anwendbar) String , der einen gültigen Pfad enthält
FileReader	File -Objekt, FileDescriptor (nur auf die Standardströme stdin, stdout, stderr anwendbar) String , der einen gültigen Pfad enthält
FileWriter	File -Objekt, FileDescriptor (nur auf die Standardströme stdin, stdout, stderr anwendbar) String , der einen gültigen Pfad enthält
FilterInputStream	InputStream
FilterOutputStream	OutputStream
FilterReader	Reader
InputStreamReader	InputStream
LineNumberReader	Reader
ObjectInputStream	InputStream
ObjectOutputStream	OutputStream
OutputStreamWriter	OutputStream
PipedInputStream	PipedOutputStream Der parameterlose Vorgabekonstruktor erzeugt einen unverbundenen Strom.
PipedOutputStream	PipedInputStream Der parameterlose Vorgabekonstruktor erzeugt einen unverbundenen Strom.
PipedReader	PipedWriter
PipedWriter	PipedReader
PrintWriter	OutputStream Writer
PrintStream	OutputStream
PushbackInputStream	InputStream
PushbackReader	Reader
SequenceInputStream	InputStream Oder Inhalt eines Objekts dessen Klasse die Enumeration -Schnittstelle implementiert.
StringReader	String

Der [LineNumberReader](#) liefert ein Beispiel eines Stroms, der auf Basis eines anderen Stroms

definiert wird. Im Falle des folgenden Beispiels wird ein `LineNumberReader` ausgehend von einem bestehenden `FileReader` erzeugt.

Wie bereits im [zweiten Kapitel angesprochen](#) unterstützt Java den Unicode-Zeichensatz. Durch ihn wird die plattformübergreifende Darstellung verschiedenster Zeichensätze ermöglicht. Als Erweiterung des klassischen ISO 8859-Teil 1 Zeichensatzes benötigt er jedoch generell 16 Bit zur Darstellung eines Symbols. Zusätzlich ist zu einem Unicode codierten Datenstrom die Codierungsschema, identifiziert durch ein eindeutiges Kürzel, anzugeben um die korrekte Darstellung zu ermöglichen.

Hierzu erlauben die Stream-Klassen [InputStreamReader](#) und [OutputStreamWriter](#) die explizite Spezifikation der Eingabe- bzw. Ausgabe Encodierung.

Jede Java-Implementierung muß mindestens folgende Code-Formate unterstützen: US-ASCII, ISO-8859-1, UTF-16BE (16-Bit Unicode im big-endian Format), UTF-16LE (dergleichen als little-endian) und UTF-16 (allgemeines 16-Bit Unicodeformat, *byte order mark* am Beginn des Stroms definiert verwendetes Anordnungsschema).

```
(1)import java.io.FileDescriptor;
(2)import java.io.FileOutputStream;
(3)import java.io.FileReader;
(4)import java.io.IOException;
(5)import java.io.LineNumberReader;
(6)import java.io.OutputStreamWriter;
(7)
(8)public class UnicodeWriter {
(9)    public static void main(String[] args) {
(10)        LineNumberReader lnr = null;
(11)        OutputStreamWriter osw = null;
(12)
(13)        String encoding, text;
(14)
(15)        try {
(16)            System.out.print("specify encoding:");
(17)            lnr = new LineNumberReader(new FileReader(FileDescriptor.
in));
(18)            encoding = lnr.readLine();
(19)
(20)            System.out.print("Specify text to encode:");
(21)            text = lnr.readLine();
(22)
(23)            System.out.print("encoded text:");
(24)
(25)            osw = new OutputStreamWriter((new FileOutputStream
(FileDescriptor.out)), encoding);
(26)            osw.write(text);
(27)        } catch (IOException ioe) {
(28)            System.out.println("an IOException occurred\n"+ioe.getMessage
());
(29)        } finally {
(30)            try {
(31)                lnr.close();
(32)                osw.close();
(33)            } catch (Exception e) {
(34)                //ignore it
(35)            } //catch
(36)        } //finally
(37)    } //main()
(38)} //class UnicodeWriter
```

Beispiel 60: Schreiben in frei wählbarem Ausgabeencoding [UnicodeWriter.java](#)

Das Programm ließt zunächst von der Standardeingabe die gewünschte Encodingdefinition und den auszugebenen Text als Zeichenkette.

Dann wird ein Ausgabestrom auf die Standardausgabe erzeugt, der das zuvor spezifizierte Encoding verwendet. Unterstützt die Java-Implementierung das angegebene Codierungsschema nicht, schlägt die Erzeugung des Ausgabestroms fehl, und es wird ein Ausnahmeereignis erzeugt. Zum Abschluß wird der Text unter Anwendung des definierten Encodingschemas über den Ausgabestrom ausgegeben.

Beispielinteraktionen:

specify encoding:US-ASCII
 Specify text to encode:abcäöüß
 encoded text:abc????

specify encoding:UTF8
 Specify text to encode:aä
 encoded text:aôçx

specify encoding:UTF-16LE
 Specify text to encode:test
 encoded text:t e s t

```
(1)import java.io.IOException;
(2)import java.io.FileDescriptor;
(3)import java.io.LineNumberReader;
(4)import java.io.FileReader;
(5)import java.io.FileWriter;
(6)
(7)
(8)public class Type {
(9)    public static void main(String[] args) {
(10)        FileReader fr = null;
(11)        FileWriter fw = null;
(12)        LineNumberReader lnr = null;
(13)        String line;
(14)
(15)        try {
(16)            fr = new FileReader(args[0]);
(17)            lnr = new LineNumberReader (fr);
(18)            fw = new FileWriter(FileDescriptor.out);
(19)
(20)            while ( (line = lnr.readLine()) != null) {
(21)                fw.write( lnr.getLineNumber()+": "+line+"\n" );
(22)            } //while
(23)        } catch (IOException ioe) {
(24)            System.out.println("an IOException occurred");
(25)            System.out.println( ioe.getMessage() );
(26)        } finally {
(27)            try {
(28)                fr.close();
(29)                fw.close();
(30)            } catch (IOException ioe) {
(31)                //ignore it
(32)            } //catch
(33)        } //finally
(34)    } //main()
(35)} //class Type
```

Beispiel 61: Zeilenweise Ausgabe einer Datei incl. Zeilennummern [Type.java](#)

Serialisierung von Objekten

Durch den Stromtyp `ObjectOutputStream` können vollständige Objekte geschrieben, und durch `ObjectInputStream` wieder in den Speicher eingeladen werden.

```
(1)import java.io.FileInputStream;
(2)import java.io.FileOutputStream;
(3)import java.io.IOException;
(4)import java.io.ObjectInputStream;
(5)import java.io.ObjectOutputStream;
(6)import java.io.Serializable;
(7)import java.util.Calendar;
(8)import java.util.GregorianCalendar;
(9)
(10)public class SerializedData {
(11)    public static void main(String[] args) {
(12)        //just for determining the current year
(13)        GregorianCalendar gregCal = new GregorianCalendar();
(14)
```

```

(15)         Person hans = new Person();
(16)         hans.name = new String("hans");
(17)         hans.yearOfBirth = 1950;
(18)         hans.age = gregCal.get(Calendar.YEAR) - hans.yearOfBirth;
(19)
(20)         System.out.println("object before serialization:\n" + hans.toString
(21)         () );
(22)
(23)         ObjectOutputStream oos = null;
(24)
(25)         try {
(26)             oos = new ObjectOutputStream( new FileOutputStream("hans") );
(27)             oos.writeObject ( hans );
(28)         } catch (IOException ioe) {
(29)             System.out.println("an IOException occurred");
(30)             System.out.println( ioe.getMessage() );
(31)         } finally {
(32)             try {
(33)                 oos.close();
(34)             } catch (IOException ioe) {
(35)                 //ignore it
(36)             } //catch
(37)         } //finally
(38)
(39)         gregCal = null;
(40)         hans = null;
(41)
(42)         ObjectInputStream ois = null;
(43)         try {
(44)             Person anotherOne;
(45)             ois = new ObjectInputStream( new FileInputStream("hans") );
(46)             anotherOne = (Person) ois.readObject();
(47)
(48)             System.out.println( "object retrieved from file:\n"+
anotherOne.toString() );
(49)         } catch (IOException ioe) {
(50)             System.out.println("an IOException occurred while reading
back object");
(51)         } catch (ClassNotFoundException cnfe) {
(52)             System.out.println("could not find class Person");
(53)         } finally {
(54)             try {
(55)                 ois.close();
(56)             } catch (IOException ioe) {
(57)                 //ignore it
(58)             } //catch
(59)         } //finally
(60)     } //main()
(61) } //class SerializeData
(62)
(63) class Person implements Serializable {
(64)     public int yearOfBirth;
(65)     public String name;
(66)     transient int age;
(67)
(68)     public void finalize() {
(69)         System.out.println("object destroyed");
(70)     } //finalize()
(71)
(72)     public String toString() {
(73)         return("name="+name+"\n"+"yearOfBirth="+yearOfBirth+"\n"+"age="+age);
(74)     } //toString()
(75) } //class Person

```

Beispiel 62: Serialisieren und Laden eines Objekts [SerializeData.java](#)

BildschirmAusgabe:

```

object before serialization:
name=hans

```

```

yearOfBirth=1950
age=50
object retrieved from file:
name=hans
yearOfBirth=1950
age=0

```

Das [transiente Attribut](#) wird nicht in die Datei übernommen. Die Inhalte aller anderen Attribute werden gesichert, und können rückgelesen werden.

Beim (Wieder-)Einlesen eines Objektes wird versucht dessen Klassendefinition aus der entsprechenden Klassendatei zu laden. Ist dies nicht möglich, so wird ein Ausnahmeereignis vom Typ [ClassNotFoundException](#) generiert.

Anmerkung: Der Java-Serialisierungsmechanismus verhindert die mehrfache Ablage desselben Objektes im Bytestrom.

Für die häufig umzusetzende Aufgabe der Eingabeformatprüfung, und anschließenden Klassifizierung in bestimmte Kategorien steht die Klasse [StreamTokenizer](#) zur Verfügung.

```

(1)import java.io.StreamTokenizer;
(2)import java.io.FileReader;
(3)import java.io.FileDescriptor;
(4)import java.io.IOException;
(5)
(6)public class TokenTest {
(7)    public static void main(String[] args) {
(8)        StreamTokenizer st = null;
(9)
(10)        int    op1=0,
(11)              op2=0;
(12)        try {
(13)            st = new StreamTokenizer(new FileReader(FileDescriptor.in));
(14)
(15)            while(st.nextToken() == StreamTokenizer.TT_NUMBER)
(16)                op1 = (op1*10) +(int) st.nval;
(17)
(18)            while(st.nextToken() == StreamTokenizer.TT_NUMBER)
(19)                op2 = (op2*10) + (int) st.nval;
(20)
(21)            System.out.println(op1+" "+op2+"="+ (op1+op2));
(22)        } catch (IOException ioe) {
(23)            System.out.println("an IOException occurred\n"+ioe.getMessage
(24)            );
(25)        } //catch
(26)} //main()
(26)} //class TokenTest

```

Beispiel 63: Einfache Addition zweier Zahlen unter Verwendung des StringTokenizers [TokenTest.java](#)

Die beiden Operanden können durch ein beliebiges nicht numerisches Zeichen voneinander abgetrennt werden, abenso kann das Berechnungsende erklärt werden.

Zum Zugriff auf **(G-)ZIP komprimierte Dateien** bieten die Klassen [ZipInputStream](#) und [GZIPInputStream](#) bzw. [ZipOutputStream](#) und [GZIPOutputStream](#) Lese- bzw. Schreibströme an.

▲ 3.2.2 Threads und Nebenläufigkeit

Java bietet mit den sog. Programmfäden engl. *Threads* die Möglichkeit an, parallele leichtgewichtige Prozesse direkt in der Hochsprache zu definieren und zu kontrollieren. Hierbei werden keine Anforderungen an eine spätere Unterstützung dieses Konzepts durch die tatsächliche physische Hardware gestellt; der gesamte Mechanismus ist rein Hochsprachen-basiert.

Als Bestandteil des [automatisch importierten](#) Paketes `java.lang` stehen Threads in jeder Applikation und jedem Applet ohne zusätzliche Aufwende zur Verfügung.

Weiterführende Informationen: [Java Language Specification, chap. 17](#) und [Java JVM Spezifikation, chap. 8](#).

Inhaltlich unterscheidet sich ein Thread nur in marginalen Modifikationen von einer gewöhnlichen Klasse. Hauptunterschied ist die eigenständige aktive und nebenläufige Ausführung von Objekte einer solchen Klasse.

Voraussetzungen zur Erzeugung eines Threads:

- Die Klasse spezialisiert die Standard-API-Klasse [Thread](#).
Wie alle Klassen, die nebenläufig ausgeführt werden sollen implementiert auch [Thread](#) die Schnittstelle [Runnable](#). Sie definiert die Operation [run\(\)](#).
- Die Klasse überschreibt die Methode [run\(\)](#).
Sie wird explizit durch die Methode [start\(\)](#) aufgerufen. [start\(\)](#) ist asynchron und kehrt sofort nach Erzeugung des Threads zum Aufrufer zurück.

Hinweis: Die Methode [run\(\)](#) sollte nicht direkt aufgerufen werden! Bei der direkten Ausführung unterbliebe die notwendige Initialisierung; insbesondere wäre der dann „gewöhnliche“ Methodenaufruf nicht asynchron, und das neue Objekt würde nicht nebenläufig ausgeführt.

```
(1)public class Threads1 {
(2)    public static void main(String[] args) {
(3)        HelloThread northGerman = new HelloThread( "Moin Moin" );
(4)        HelloThread southGerman = new HelloThread( "Gruess Gott" );
(5)
(6)        northGerman.start();
(7)        southGerman.start();
(8)    } //main()
(9)} //class Threads
(10)
(11)class HelloThread extends Thread {
(12)    protected String greetingText;
(13)
(14)    public HelloThread (String greetingText) {
(15)        this.greetingText = greetingText;
(16)    } //constructor
(17)
(18)    public void run() {
(19)        while (true) {
(20)            try {
(21)                Thread.sleep(500);
(22)            } catch (InterruptedException ie) {
(23)                System.out.println("an InterruptedException occurred
\n"+ie.toString()+"\n"+ie.getMessage() );
(24)            } //catch
(25)            System.out.println( greetingText);
(26)        } //while
(27)    } //run()
(28)}//class HelloThread
```

Beispiel 64: Zwei einfache Threads [Threads1.java](#)

Das Programm gibt die beiden Texte *Moin Moin* und *Gruess Gott* jeweils im Wechsel durch separate Threads aus.

Besonders fällt auf, daß die Applikation nach Abarbeiten der letzten Anweisung der main-Methode nicht terminiert.

Java-Programme die (noch) *Vordergrund*-Threads ausführen terminieren nicht am Ende der main-Methode, sondern führen weiterhin die noch laufenden Threads -- bis zu deren eigenständigem Terminieren oder Abbruch -- aus.

Die zweite Threadklasse bilden die [Daemon-Threads](#). Verfügt ein laufendes Programm ausschließlich über solche Hintergrund-Threads terminiert es am Ende der main-Methode wie gewohnt.

Demnach läßt sich der bisher bekannte Programmtyp als nebenläufige Applikation mit dem Vordergrundthread main betrachten. Terminiert dieser Thread, so terminiert auch die gesamte Applikation.

Hinweis: Programme mit laufenden Vordergrundthreads lassen sich durch Beenden der virtuellen

Maschine mittels `System.exit(int)` terminieren.

Das Beispiel enthält auch bereits zwei der möglichen Status innerhalb des Lebenszyklus eines Threads, nämlich *erzeugt* (aber noch nicht in Ausführung befindlich, nach Aufruf des Konstruktors) und *in Ausführung* nach dem Starten durch den Aufruf der Methode `run()`.

Die möglichen Threadzustände können jedoch noch durch weitere Methoden beeinflusst werden:

- `start()`
Erzeugung eines Threads (starten) aus einem bereits erstellten Thread-Objekts.
- `sleep(long)` und `sleep(long, int)`
Anhalten eines Threads für eine durch die Übergabeparameter festgelegte Zeitspanne. Während der Ausführungspause bleiben gesetzte Sperren (Monitore) erhalten.
- `join()`, `join(long)` und `join(long, int)`
Wartet auf das Terminieren des erzeugten Threads. (Quasi synchrone Variante von `run()`). Optional kann eine maximale abzuwartende Zeitspanne definiert werden.
- `yield`
Freiwillige Abgabe der Zeitscheibe.
- `destroy()`
Zerstört Thread ohne vorherige Aufräumarbeiten; insbesondere werden gesetzte Sperren nicht freigegeben.

Hinweis:

Die beiden `stop`-Methoden, sowie die Methoden `suspend` und `resume` sind seit der Version 1.3 als *deprecated* gekennzeichnet und sollten nicht mehr verwendet werden!

Der Grund liegt in der, mit der durch die API zugesicherten, sofortigen Unterbrechung. Dabei kann die Unterbrechung auch innerhalb eines kritischen Abschnittes erfolgen, wodurch es zu verschiedensten Inkonsistenzen und weiteren Folgeproblemen kommen kann.

Hinzu kommen von `java.lang.Object` ererbte Methoden:

- `wait()`
`wait(long)`
`wait(long, int)`
Hält Thread beim Zugriff auf ein bestimmtes Objekt an, bis ein anderer Thread die Methode `notify()` oder `notifyAll()` ausführt. Zur Synchronisation werden Monitore verwendet.
- `notify()`
`notifyAll()`
Führt mit der Ausführung eines im Threads fort, der beim Zugriff auf ein gesperrtes Objekt in den Wartezustand versetzt wurde.

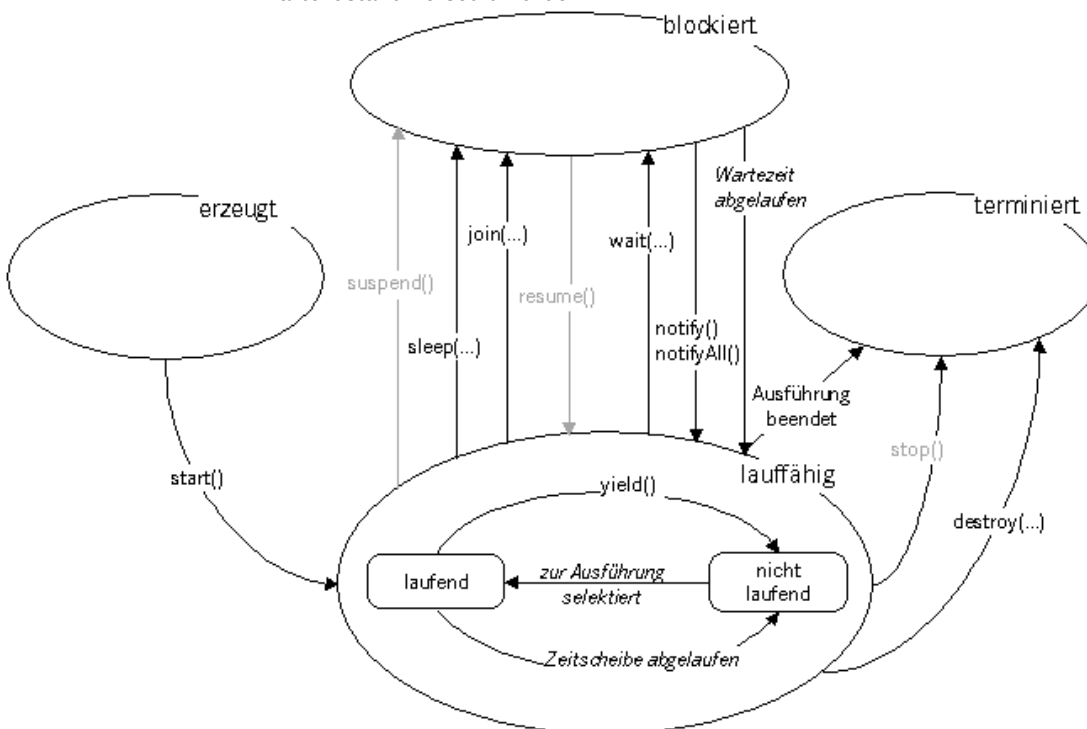


Abbildung 53: Thread-Zustände (click on image to enlarge!)

Die Graphik zeigt die verschiedenen möglichen Zustände eines Threads. Als *deprecated* gekennzeichnete Methoden sind grau unterlegt.

Das interne Scheduling aller laufenden Threads erfolgt prioritätsgesteuert. Die **Prioritäten** zugreifbarer (d.h. eigen erzeugter) Threads kann erfragt und gezielt verändert werden. Die Threadpriorität wird als ganzzahliger Wert angegeben. Plattformspezifisch kann dieser Wert variieren; jedoch kann die konkrete Unter- und Obergrenze über die Konstanten [MIN_PRIORITY](#) bzw. [MAX_PRIORITY](#) zur Laufzeit erfragt werden. Ohne manuellen Eingriff werden neue Threads mit der durch [NORM_PRIORITY](#) definierten Priorität erzeugt.

Methoden zur Beeinflussung der Thread-Priorisierung:

- [int getPriority\(\)](#)
liefert die Priorität des Threads.
- [setPriority\(int\)](#)
versucht die Priorität auf den übergebenen Wert zu setzen. Dies kann wegen mangelnder Berechtigung, oder ungültiger Übergabeparameter, fehlschlagen. In diesem Falle wird ein Ausnahmeereignis der Klasse [SecurityException](#), bzw. -- bei ungültigen Übergabeparametern -- [IllegalArgumentException](#) erzeugt.

Aus Gründen der vereinfachten Verwaltung können Threads in Gruppen (API-Klasse [ThreadGroup](#)) zusammengefaßt und damit gebündelt beeinflußt werden.

Operationen zur Zustandsänderung die auf einzelnen Threads wirken, können auch auf **Thread-Gruppen** angewendet werden.

Die Verwaltung von solchen Thread-Bündeln ist in der API-Klasse [ThreadGroup](#) zusammengefaßt; die Gruppenzugehörigkeit eines spezifischen Threads kann durch [getThreadGroup\(\)](#) ermittelt werden.

Daemon-Threads:

wie bereits angerissen gibt es neben den „normalen“ Anwenderthreads auch die Klasse der Daemon-Threads.

Ihr Hauptunterscheidungsmerkmal zu den Anwenderthreads ist das Charakteristikum, daß die JVM auch terminiert, wenn sich Threads dieser Kategorie in Ausführung befinden. Dies prädestiniert sie für Hintergrundaufgaben wie Verwaltungs- oder Überwachungstätigkeiten.

Der Typ eines Threads kann vor dessen Ausführungsbeginn durch den `start()`-Aufruf mittels der Methode [setDaemon\(boolean\)](#) festgelegt, und zur Laufzeit mit [isDaemon\(\)](#) jederzeit während der Ausführung erfragt, werden.

```
(1)public class IsPrime {
(2)    public static void main(String[] args) {
(3)        PrimitivePrimeTest ppt;
(4)        StatusInformation myInfo = new StatusInformation();
(5)        myInfo.setDaemon(true); //declare as daemon thread
(6)        myInfo.setPriority( (Thread.NORM_PRIORITY + 2) <= Thread.
MAX_PRIORITY ? Thread.NORM_PRIORITY + 2 : Thread.MAX_PRIORITY );
(7)        myInfo.start();
(8)
(9)        ThreadGroup workerThreads      = new ThreadGroup("worker threads");
(10)
(11)        for (int i = Integer.parseInt(args[0]); i <= Integer.parseInt(args
[1]); i++) {
(12)            ppt = new PrimitivePrimeTest(workerThreads, i, "calculating
" + i );
(13)            ppt.start();
(14)        } //for
(15)
(16)    } //main
(17)} //class IsPrime2
(18)
(19)class PrimitivePrimeTest extends Thread {
(20)    protected int numberToTest;
(21)    boolean earlyExit=false;
(22)
(23)    public PrimitivePrimeTest(ThreadGroup tg, int numberToTest, String
threadName) {
(24)        super (tg, threadName); //call to super's class constructor
(25)        this.numberToTest = numberToTest;
(26)    } //constructor
(27)
(28)    public void run() {
(29)        if (numberToTest == 1)
(30)            earlyExit = true;
(31)        for (int i=2; i < ((int) Math.sqrt(numberToTest))+1; i++) {
(32)            try {
```

```

(33)                Thread.sleep(1000);
(34)                } catch (InterruptedException ie) {
(35)                //ignore it
(36)                } //catch
(37)                if (numberToTest % i == 0) {
(38)                    earlyExit=true;
(39)                    break;
(40)                } //endif
(41)            } //for
(42)            if (earlyExit != true)
(43)                System.out.println(numberToTest+" is prime");
(44)        } //run()
(45)} //class PrimitivePrimeTest
(46)
(47)class StatusInformation extends Thread {
(48)    public void run() {
(49)        while (true) {
(50)            System.out.println("no of currently running threads:
"+Thread.activeCount() );
(51)            try {
(52)                Thread.sleep(500);
(53)            } catch (InterruptedException ie) {
(54)                //ignore it
(55)            } //catch
(56)        } //while
(57)    } //run()
(58)} //class StatusInformation

```

Beispiel 65: Ein einfacher Primzahlenprüfer [IsPrime.java](#)

Beispielablauf:

```

$java IsPrime 1 25
no of currently running threads: 2
2 is prime
3 is prime
no of currently running threads: 24
no of currently running threads: 24
5 is prime
7 is prime
no of currently running threads: 11
no of currently running threads: 11
11 is prime
13 is prime
no of currently running threads: 6
no of currently running threads: 6
17 is prime
19 is prime
23 is prime
no of currently running threads: 3
no of currently running threads: 3

```

Das Programm prüft in jeweils einem eigenen Thread, ob die Ganzzahlen im Intervall zwischen den gegebenen Grenzen prim sind. Entdeckte Primzahlen werden mit einer entsprechenden Meldung ausgegeben.

Um die unterschiedliche Ausführungsdauer der jeweiligen Threads hervorzuheben führt jeder Programmfaden nur eine Berechnung pro Sekunde aus.

Die Berechnungsthreads sind alle in einer eigenen Threadgruppe (workerThreads) zusammengefaßt. Jeder Thread innerhalb dieser Gruppe wird durch die Zeichenkette *calculating* gefolgt von der zu prüfenden Zahl benannt.

Zusätzlich gibt ein Daemon-Thread alle halbe Sekunde die Anzahl der (noch) aktiven Threads aus. Dieser Thread wird automatisch durch die JVM nach dem Abarbeiten des letzten aktiven Anwenderthreads terminiert. Die Priorität des Daemon-Threads ist um zwei gegenüber dem Vorgabewert erhöht, sofern dadurch nicht die maximal erlaubte Priorisierung überschritten wird.

Synchronisation von Methodenzugriffen:

```

(1)import java.io.DataOutputStream;
(2)import java.io.FileOutputStream;
(3)import java.io.FileInputStream;
(4)import java.io.DataInputStream;
(5)import java.io.IOException;
(6)import java.io.FileNotFoundException;
(7)
(8)public class Threads2 {
(9)    public static void main(String[] args) {
(10)        for (int threadCount=0; threadCount < Integer.parseInt(args[0]);
threadCount++)
(11)            (new IncrementCounter()).start();
(12)    } //main()
(13)} //class Threads2
(14)
(15)class IncrementCounter extends Thread {
(16)    public void run() {
(17)        int counterValue;
(18)        DataInputStream dis = null;
(19)        DataOutputStream dos = null;
(20)
(21)        try {
(22)            for (int i=0; i<10; i++) {
(23)                dis = new DataInputStream( new FileInputStream
( "counter" ) );
(24)                counterValue = dis.readInt();
(25)                dis.close();
(26)
(27)                System.out.println(counterValue+" read by thread "+
(Thread.currentThread()).getName() );
(28)
(29)                dos = new DataOutputStream( new FileOutputStream
( "counter", false ) );
(30)                dos.writeInt( ++counterValue );
(31)                dos.close();
(32)            } //for
(33)        } catch (FileNotFoundException fnfe) {
(34)            System.out.println("file counter could not be opened!
\n"+fnfe.toString()+"\n"+fnfe.getMessage() );
(35)            System.exit(1);
(36)        } catch (IOException ioe) {
(37)            System.out.println("an IOException occurred in thread
"+(Thread.currentThread()).getName()+"!\n"+ioe.toString()+"\n"+ioe.getMessage() );
(38)            System.exit(1);
(39)        } //catch
(40)    } //run()
(41)} //class IncrementCounter

```

Beispiel 66: Gemeinsames Inkrementieren eines Zählers in einer Datei [Threads2.java](#)

Hilfsprogramme: [manuelles Rücksetzen des Zählerstandes auf 0](#), [manuelles Auslesen des Zählerstandes und Ausgabe auf Standardausgabe](#)

Das Programm liest den Stand einer ganzzahligen Variable aus einer Datei aus, erhöht um Eins und schreibt ihn zurück in dieselbe Datei, unter Verlust des alten Standes (Überschreiben). Die beschriebene Vorgangsfolge wird im Multithreading-Betrieb durch mehrere Ausführungsfäden nebenläufig durchgeführt.

Beim Auffangen eines Ausnahmeereignisses wird die virtuelle Maschine terminiert, da das einzelne Terminieren nur eines Threads das Ergebnis verfälschen würde.

Tritt zwischen dem Auslesevorgang aus der Datei, und dem Rückschreiben des modifizierten Wertes ein Threadwechsel ein, so kommt es zum Phänomen des *lost updates*.

mögliche Bildschirmausgaben:

Nach dem Einlesen der Zahl 5 durch Thread-33 tritt, vor ihrem inkrementierten Rückschreiben, der Threadwechsel ein, wodurch der nachfolgend ausgeführte Thread-34 dieselbe Zahl nochmals ausliest.

```

$java Threads2
...

```

```

3 read by thread Thread-28
4 read by thread Thread-30
5 read by thread Thread-33
5 read by thread Thread-34
6 read by thread Thread-42
7 read by thread Thread-45
...

```

Sicherlich ein Extremum dieses Verhaltens stellt das nachfolgende Beispiel dar:

mögliche Bildschirmausgaben:

Threadwechsel tritt jeweils direkt nach dem Einlesen auf. Als Konsequenz wird derselbe Zahlenwert mehrfach durch verschiedene Threads gelesen.

```

$java threads2
...
10 read by thread Thread-35
10 read by thread Thread-10
10 read by thread Thread-36
10 read by thread Thread-37
10 read by thread Thread-14
10 read by thread Thread-38
10 read by thread Thread-12
10 read by thread Thread-39
10 read by thread Thread-16
10 read by thread Thread-41
10 read by thread Thread-17
10 read by thread Thread-19
10 read by thread Thread-42
10 read by thread Thread-18
10 read by thread Thread-40
10 read by thread Thread-0
10 read by thread Thread-11
...

```

Um Daten-Inkonsistenzen zur Laufzeit zu verhindern, wie sie potentiell entstehen könnten, wenn zwei Programmfäden gleichzeitig dieselbe Methode ausführen oder zeitlich verschränkt dieselbe kritische Ressource benutzen, ist mit dem (aus Kapitel zwei bekannten) Schlüsselwort [synchronized](#) ein Hochsprachenmechanismus gegeben um mehrere Aufrufer gezielt zu serialisieren.

Technisch handelt es sich dabei um das aus den Betriebssystemen bekannte Konzept der Monitore (siehe [C. A. R. Hoare: Monitors: An Operating System Structuring Concept](#)) zur Synchronisation des Zugriffs auf kritische Abschnitte.

Jedoch besteht bei dieser Vorgehensweise die Gefahr eines **Deadlocks** durch wechselseitiges Blockieren!

Durch synchronisierte Blöcke können nicht nur einzelne Methoden, sondern Bündel von Zugriffen gemeinsam geschützt werden.

Hierzu wird dem Schlüsselwort `synchronized` ein auswertbarer Ausdruck nachgestellt, der die Resource bezüglich der zu synchronisieren ist bezeichnet.

Im Falle des betrachteten Beispiels ist daher hinsichtlich der gemeinsam beanspruchten (und daher kritischen) Ressource der Zählerstandsdatei zu synchronisieren.

```

(1)import java.io.DataInputStream;
(2)import java.io.DataOutputStream;
(3)import java.io.File;
(4)import java.io.FileInputStream;
(5)import java.io.FileNotFoundException;
(6)import java.io.FileOutputStream;
(7)import java.io.IOException;
(8)
(9)public class Threads21 {
(10)    public static synchronized void main(String[] args) {
(11)        File counterFile = new File ("counter");
(12)
(13)        for (int threadCount=0; threadCount < Integer.parseInt(args[0]);
threadCount++)
(14)            (new IncrementCounter(counterFile)).start();
(15)    } //main()
(16)} //class Threads21

```

```

(17)
(18)class IncrementCounter extends Thread {
(19)    File    counterFile;
(20)    public IncrementCounter(File rwFile) {
(21)        counterFile = rwFile;
(22)    } //constructor
(23)
(24)    public void run() {
(25)        int counterValue;
(26)        DataInputStream dis = null;
(27)        DataOutputStream dos = null;
(28)
(29)        try {
(30)            for (int i=0; i<10; i++) {
(31)                synchronized (counterFile) {
(32)                    dis = new DataInputStream( new
FileInputStream( counterFile ));
(33)                    counterValue = dis.readInt();
(34)                    dis.close();
(35)
(36)                    System.out.println(counterValue+" read by
thread "+ (Thread.currentThread()).getName() );
(37)
(38)                    dos = new DataOutputStream( new
FileOutputStream( counterFile ));
(39)                    dos.writeInt( ++counterValue );
(40)                    dos.close();
(41)                } //synchronized
(42)            } //for
(43)        } catch (FileNotFoundException fnfe) {
(44)            System.out.println("file counter could not be opened!
\n"+fnfe.toString()+"\n"+fnfe.getMessage() );
(45)            System.exit(1);
(46)        } catch (IOException ioe) {
(47)            System.out.println("an IOException occurred in thread
"+(Thread.currentThread()).getName()+"!\n"+ioe.toString()+"\n"+ioe.getMessage() );
(48)            System.exit(1);
(49)        } //catch
(50)    } //run()
(51)} //class IncrementCounter

```

Beispiel 67: Lost-Update-freie Variante des vorhergehenden Beispiels [Threads21.java](#)

Auffallendstes Kennzeichen der Synchronisation mittels Schlüsselwort `synchronized` ist es, daß gemeinsam genutzte Objekte existieren müssen, um die alle Threads konkurrieren. Auf den Einsatzfall einer eher losen Kopplung der einzelnen Ausführungseinheiten sind die Methoden `wait` und `notify` ausgelegt. Der wohl bekannteste Vertreter dieser Problemklasse ist das klassische *Erzeuger-Verbraucher-Schema*, in dem zwei grundlegend verschiedene Rollen, die des Erzeugers und die des zeitlich nachgelagerten -- und daher zu synchronisierenden -- Verbrauchers unterschieden werden. Für diese Problemklasse bietet Java das Schlüsselwort `wait` an. Es erlaubt das Abwarten eines Ereignisses, welches durch `notify` angezeigt wird.

Das Standardschema zur Benutzung lautet:

```

synchronized doWhenCondition() {
    while (!condition)
        wait();
    ...Bedingung true...
} //synchronized

```

Als Randbedingung gilt die zwingende Einbettung in eine als `synchronized` deklarierte Methode, um die Modifikation der While-Bedingung durch nebenläufig ausgeführte Threads zu verhindern. Der `wait`-Aufruf suspendiert die Ausführung des Prozesses, und gibt gleichzeitig (atomar) seine gesetzten Sperren frei.

Das Benutzungsschema für `notify` lautet:

```

synchronized changeCondition() {

```

```

        ...Änderung von Werten, die in der Bedingung auftreten...
        notify();
    } //synchronized

```

Die wait-Methode steht in verschiedenen Ausgestaltungen zur Verfügung:

[wait\(\)](#) wartet bis zur Wiedererweckung durch [notify](#); gleiche Wirkung wie [wait\(0\)](#).

[wait\(long\)](#) wartet bis zum Ablauf des durch den Übergabeparameter fixierten Zeitraumes in Millisekunden auf den Aufruf von [notify\(\)](#).

[wait\(long, int\)](#) wartet bis zum Ablauf des durch die Übergabeparameter spezifizierten Zeitraumes -- der long-Wert wird als Millisekunden interpretiert, zu dem die als int-Wert gegebenen Nannosekunden addiert werden -- auf den Aufruf von [notify\(\)](#).

Zur Wiedererweckung wartender Threads kann die parameterlose Methode [notify\(\)](#) benutzt werden; sie erweckt maximal einen wartenden Thread.

Den Wiederanlauf beliebig vieler wartender Threads ermöglicht [notifyAll\(\)](#).

Anmerkungen:

- Konstruktoren sind implizit *synchronized*, da ein Objekt nur genau von einer aktiven Einheit gleichzeitig erzeugt werden kann.
- Synchronisierte Klassenmethoden ziehen eine Sperre auf Klassenebene nach sich.
- synchronized-Angaben können in erbenden Klassen überschrieben werden. Das Verhalten der Superklasse bleibt hiervon jedoch unberührt.
- Synchronisierte Abschnitte erfordern zusätzlichen Byte-Code zur Zugriffskontrolle und notwendigen Serialisierung, daher sinkt durch Nutzung dieses Mechanismus im Allgemeinen die Ausführungsgeschwindigkeit.

Ermitteln von Threadinformation:

Die Anzahl der laufenden aktiven Threads der aktuellen Gruppe kann durch [activeCount\(\)](#) ermittelt werden.

Informationen über jeden Thread der virtuellen Maschine liefern die Methoden [getName\(\)](#) und [setName\(String\)](#). Diese beiden Methoden erlauben die Abfrage, bzw. Definition eines Namens für jeden Thread. Bei der Erzeugung eines Threads wird automatisch ein Name durch die virtuelle Maschine vergeben. (SUNs JVM setzt hier die Zeichenkette Thread-*i*, wobei *i* die laufende Nummer des Threads ist.)

[enumerate\(Thread\[\]\)](#) befüllt den als Parameter übergebenen Array mit Verweisen auf die derzeit aktiven Thread Objekte.

Der aktuelle Thread kann durch die statische Methode [currentThread\(\)](#) ermittelt werden.

Außer durch Spezialisierung der Klasse Thread kann durch Impelementierung des **[Interfaces Runnable](#)** Nebenläufigkeit erzeugt werden.

Die Schnittstelle definiert ausschließlich die Operation run. Auch die Umsetzung der Klasse Thread nutzt in den vordefinierten Konstruktoren dieses Interface. Durch den Methodenaufruf [start\(\)](#) wird im zugehörigen nebenläufigen Objekt per *o.run()* (*o* ist hierbei vom Typ [Runnable](#)) die Ausführung begonnen.

Hinweis: Dieser Anwendungsfall ist insbesondere für Applets von Bedeutung, da hier keine Spezialisierung von Thread erfolgen kann, da zwingend von Klasse [Applet](#) abgeleitet werden muß. ([Beispiel dazu](#))

[Mehr zu Java Threads erfahren ... \(eigene Vorlesung zum Thema\)](#)

3.2.3 Applets

Applets sind Java-Programme die innerhalb einer Browserumgebung ausgeführt werden. Daher gelten für sie einige besondere Charakteristika, die sie von den bisher betrachteten Java-Applikationen unterscheiden:

- Ein Applet muß innerhalb einer [\(X\)HTML](#)-Seite referenziert werden, es kann nicht eigenständig ausgeführt werden.
- Die Applet-Hauptklasse muß zwingend von [Applet](#) abgeleitet sein.
- Ein Applet verfügt über keine main-Methode. Stattdessen erzeugt der Browser ein Objekt der Hauptklasse und führt die Methoden [init](#) und [start](#) automatisch aus.

- Durch die Browereinbettung, konkret: dessen [Sicherheitskonzept](#), ist es einem Applet nicht gestattet auf die lokalen Ressourcen (Dateisystem, etc.) des Rechners zuzugreifen. Als Ausnahme hierzu können spezielle „vertrauenswürdige“ Applets erstellt werden, die durch den Programmierer [signiert](#) sein müssen.
- Applets bieten keine Textschnittstelle wie (Kommandozeilen-)Applikationen. Daher muß zwingend eine graphische Benutzeroberfläche erstellt werden.

Hinweis: Zu Testzwecken kann ein Applet auch durch einen Applet-Viewer, wie der u.a. [in SUNs JDK enthalten](#) ist, ausgeführt werden. Hierfür wird neben dem Applet selbst eine minimale HTML-Seite benötigt.

Die Klassenhierarchie von Applet und ihre Klassen:

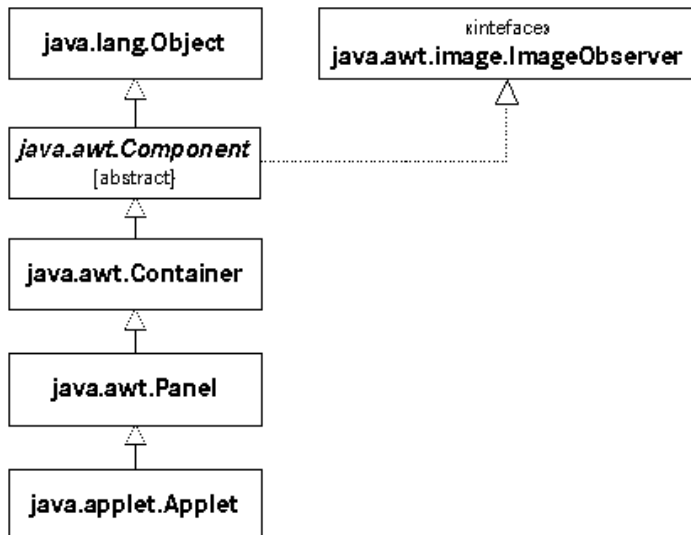
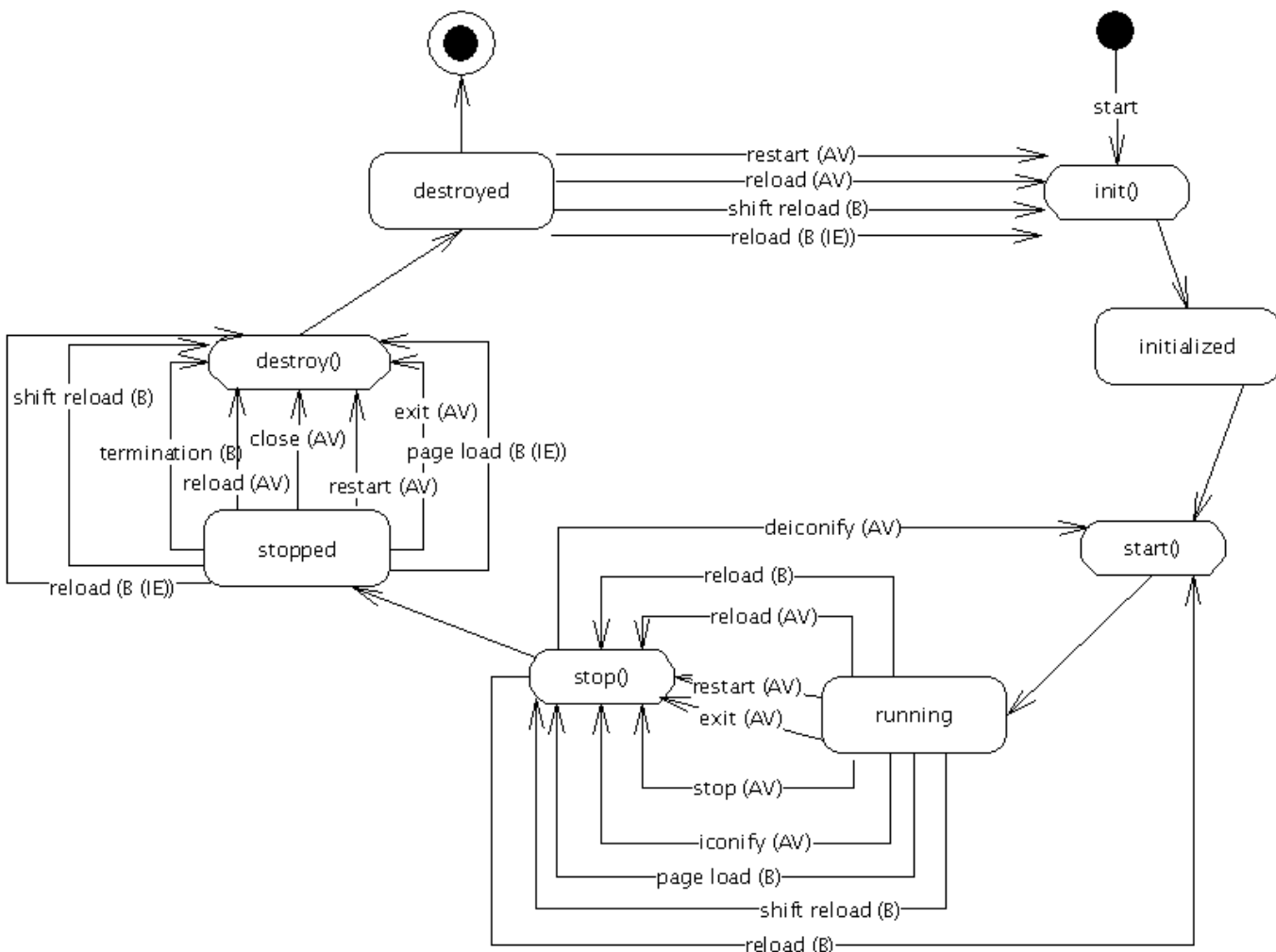


Abbildung 54: Klassenhierarchie der oberhalb der Klasse Applet (click on image to enlarge!)

- Panel: einfachste Container-Klasse. Sie stellt Platz zum Einhängen beliebiger weiterer Komponenten zur Verfügung.
- Container: Generischer Container des *Abstract Window Toolkit* (AWT), der andere AWT-Komponenten enthalten kann.
- Component: Objekt das über eine graphische Repräsentation verfügt, die am Bildschirm angezeigt werden kann, und die die Möglichkeit zur Benutzerinteraktion bietet.
Beispiele: Schaltflächen (Button), Check Boxen und Scrollbars.

Lebenszyklus eines Applets -- die **zentralen Methoden der Klasse Applet**:

- [init\(\)](#)
Durch den Browser nach dem Laden des Applets, jedoch noch vor Beginn seiner Ausführung, aufgerufen.
Verwendung: Parameter einlesen, Laden von Bildern und Audiodateien, Objekterzeugung.
- [start\(\)](#)
Direkt nach Abschluß der `init()`-Methode durch den Browser aktiviert. Darüberhinaus wird diese Methode immer dann aufgerufen, wenn die beherbergende Web-Seite neu geladen wird.
Verwendung: Starten von Animationen, Audiowiedergabe.
- [stop\(\)](#)
Durch den Browser aufgerufen, wenn die beherbergende Seite nicht mehr angezeigt wird.
Verwendung: Beenden von Multimediewiedergaben.
- [destroy\(\)](#)
Im Anschluß an `stop()` durch den Browser aufgerufen, um die durch das Applet belegten Systemressourcen zurückzufordern.
Verwendung: Explizite Speicher-Freigabe (z.B. [flush](#)-Methode für Images).



Das Zustandsübergangsdiagramm zeigt die verschiedenen Status im Leben eines Applets. Die Übergänge sind mit den auslösenden Ereignissen beschriftet. Ist kein Ereignis angetragen, so erfolgt der Übergang automatisch ohne weitere Bedingungen. In Klammern ist die jeweils ereignisauslösende Umgebung (AV für Appletviewer und B für Web-Browser) angegeben. Im Allgemeinen verhalten sich die beiden großen Browser gleich. Weicht das Verhalten eines Browsers ab, so ist dies explizit angegeben (IE kennzeichnet hierbei den MS Internet Explorer).

- [paint\(Graphics\)](#)
Wird zum vollständigen Neuzeichnen des Applets aufgerufen.
- [update\(Graphics\)](#)
Aktualisiert Fensterinhalt.

Die **benötigten (X)HTML-Strukturen:**

Zusätzlich können (gemäß HTML v4.01 Standard) noch weitere Attribute (Namen-Wert-Paare wie "code" oder "width") definiert werden:

archive -- ZIP komprimiertes Archiv das Applet gesucht wird.

name -- eindeutige Benennung des Applets innerhalb der beherbergenden Seite.

hspace -- horizontaler Abstand des Applets zur Umgebung.

vspace -- vertikaler Abstand zur Umgebung.

Ferner existieren noch einige proprietäre Attribute, die jedoch nicht durch alle Browser unterstützt werden.

Parameterübergabe an Applets:

erfolgt durch Name-Wert-Paare der Form `<param name="..." value="...">`, die in den Grenzen der Applet-Tags eingeschlossen sind.

```
<applet code="HelloWorldApplet.class" width="100" height="100">
  <param name="color" value="green">
</applet>
```

Mit XHTML v1.0, dem Nachfolger des HTML v4.01 Standards, wurde die Applet-Syntax für veraltet (*deprecated*) erklärt, und nunmer ausschließlich die Object-Variante zugelassen. Über die Änderung der Tag-Namen hinausgehend dürfte die Platzierung der Referenz auf die Java-Klassendatei, und die explizite Typangabe, als Parameter die augenfälligste Änderung sein. Diese Syntaxvariante wird jedoch, obgleich Standard, noch nicht von allen derzeit gängigen Browservarianten unterstützt.

```
<object>
  <param name="type" value="application/x-java-applet">
  <param name="JAVA_CODE" value="HelloWorldApplet.class">
```

Hinweis: In älteren Referenzen findet sich teilweise noch die ursprüngliche HotJava-Syntax, welche ein `app`-Tag definiert. Ferner enthält dieses die Referenz auf die Klasse als Attribut, jedoch ohne die Dateierweiterung `.class` explizit anzuführen.

Das Hello World-Applet:

Das absolute Minimalapplet, es schreibt lediglich den bekannten Schriftzug, kann folgendermaßen umgesetzt werden:

```
(1)import java.applet.Applet;
(2)import java.awt.Graphics;
(3)
(4)public class HelloWorldApplet extends Applet {
(5)
(6)    public void paint(Graphics g) {
(7)        g.drawString(    getParameter("displayText"),
(8)                                Integer.parseInt(getParameter
("xPos")),
(9)                                Integer.parseInt(getParameter
("yPos"))) );
(10)    } //paint()
(11)    public String getAppletInfo() {
(12)        return( "written by Mario Jeckle\nversion: 1.1\nCopyright (c) by
Mario Jeckle");
(13)    } //getAppletInfo()
(14)    public String[][] getParameterInfo() {
(15)        String appInfo[][] = {
(16)            {"displayText", "anyString", "string to display"},
(17)            {"xPos", "numeric value" , "horizontal position"},
(18)            {"yPos", "numeric value", "vertical position"}
(19)        };
(20)        return appInfo;
(21)    } //getParameterInfo()
(22)} //end class HelloWorldApplet
```

Beispiel 68: Hello World als Java-Applet [HelloWorldApplet.java](#)

[Applet ausführen](#)

Interaktion mit der Ausführungsumgebung

Durch verschiedene vordefinierte Methoden und Schnittstellen kann jedes Applet mit seiner Ausführungsumgebung, dem Browser oder Applet Viewer, in Interaktion treten. Hierdurch können von Seiten des Applets auch Dienste wie Caching, Anzeigen einer Nachricht, Abspielen von Audiodateien der Ausführungsumgebung benutzt werden.

Einfachster Fall ist die Methode [getAppletInfo](#). Sie liefert eine Zeichenkette zurück, welche üblicherweise Informationen über den Autor, die Version und die Rechte am Applet beinhaltet.

Eine ähnliche Funktion erfüllt die Methode [getParameterInfo\(\)](#). Sie liefert einen Array von

dreielementigen String-Arrays zurück, welche die erlaubten Parameter näher beschreiben.

Mittels `getParameter(String)` können die Werte jedes einzelnen Parameters ausgelesen werden.

```
(1)import java.applet.Applet;
(2)import java.awt.Graphics;
(3)
(4)public class HelloWorldApplet extends Applet {
(5)
(6)    public void paint(Graphics g) {
(7)        g.drawString(    getParameter("displayText"),
(8)                                Integer.parseInt(getParameter
("xPos")),
(9)                                Integer.parseInt(getParameter
("yPos"))) );
(10)    } //paint()
(11)    public String getAppletInfo() {
(12)        return( "written by Mario Jeckle\nversion: 1.1\nCopyright (c) by
Mario Jeckle");
(13)    } //getAppletInfo()
(14)    public String[][] getParameterInfo() {
(15)        String appInfo[][] = {
(16)            {"displayText", "anyString", "string to display"},
(17)            {"xPos", "numeric value", "horizontal position"},
(18)            {"yPos", "numeric value", "vertical position"}
(19)        };
(20)        return appInfo;
(21)    } //getParameterInfo()
(22)} //end class HelloWorldApplet
```

Beispiel 69: Parametrisierung des bekannten HelloWorld-Applets [HelloWorldApplet.java](#)

Applet ausführen

Die Erweiterung des bekannten Beispielapplets zeigt den Einsatz der `getAppletInfo`-Methode zur Rückgabe Applet-spezifischer Inhalte.

Zusätzlich wird der anzuzeigende Text und seine Position (=Parameter der Methode `drawString`) über Parameter gesteuert, die durch die Methode `getParameterInfo` in ihrer Funktion näher beschrieben sind.

Die Parameter werden aus der [\(X\)HTML-Quelle](#) durch die Methode `getParameter` ausgelesen.

Die Schnittstelle [AppletContext](#) erlaubt die aktive Beeinflussung der Ausführungsumgebung des Applets. Die Methode `getAppletContext` liefert die aktuelle Context-Ausprägung zurück.

Jedes `AppletContext`-Objekt stellt folgende Methoden zur Verfügung:

- [getApplet\(String\)](#)
liefert ein Applet-Objekt zurück, das entsprechend der übergebenen Zeichenkette [benannt](#) ist.
- [getApplets\(\)](#)
Liefert alle Applets im aktuellen Kontext (d.h. dem aktuell angezeigten (X)HTML-Dokument) zurück.
- [getAudioClip\(URL\)](#)
Liefert ein durch URL identifiziertes [AudioClip](#)-Objekt zurück.
- `getImage(URL)`
Liefert eine über URL identifizierte Graphik als [Image](#)-Objekt zurück.
- [showDocument\(URL\)](#) und [showDocument\(URL, String\)](#)
Laden das durch die URL bezeichnete Dokument, bzw. den bezeichneten Frame, in die aktuelle Ausführungsumgebung, sofern es sich dabei um einen Browser handelt. Appletviewer ignorieren diesen Methodenaufruf.
- [showStatus\(String\)](#)
Zeigt die übergebene Zeichenkette im Statusfenster des Browsers oder Appletviewers an.

```
(1)import java.applet.Applet;
(2)import java.awt.Graphics;
(3)import java.net.URL;
(4)import java.net.MalformedURLException;
(5)
(6)public class HWAURLForward extends Applet {
(7)    public void paint(Graphics g) {
(8)
(9)        g.drawString(    getParameter("displayText"),
(10)                                Integer.parseInt(getParameter
("xPos")),
```

```

(11)                                     Integer.parseInt(getParameter
("yPos")) );
(12)         try {
(13)             Thread.sleep(5000); //five seconds delay
(14)         } catch (InterruptedException iee) {
(15)             System.out.println("an InterruptedException occurred");
(16)         } //catch
(17)
(18)         try {
(19)             (this.getAppletContext()).showDocument( new URL(getParameter
("URL")) );
(20)         } catch (MalformedURLException murle) {
(21)             System.out.println("illegal URL");
(22)         } //catch
(23)     } //paint()
(24)     public String getAppletInfo() {
(25)         return( "written by Mario Jeckle\nversion: 1.1\nCopyright (c) by
Mario Jeckle");
(26)     } //getAppletInfo()
(27)     public String[][] getParameterInfo() {
(28)         String appInfo[][] = {
(29)             {"displayText", "anyString", "string to display"},
(30)             {"xPos", "numeric value" , "horizontal position"},
(31)             {"yPos", "numeric value", "vertical position"},
(32)             {"URL", "any valid URL according to IETF's RFC 1738", "URL
to jump to"}
(33)         };
(34)
(35)         return appInfo;
(36)     } //getParameterInfo()
(37)} //class HWAURLForward

```

Beispiel 70: Hello World Applet mit URL-Forward nach fünf Sekunden [HWAURLForward.java](#)

Das Beispiel erweitert das parametrisierte HelloWorld-Applet um eine URL-Weiterleitung nach fünf Sekunden. Die notwendige Zieladresse wird aus dem Parameter URL geladen.

[zugehörige HTML-Seite](#)

Sicherheitskonzept von Applets:

Nachfolgende Tabelle faßt die Rechte und Beschränkungen von Applikationen und Applets, sowohl für die Ausführungsumgebung Web-Browser als auch im Appletviewer, zusammen.

Aktion	Browser	Applet Viewer	Java Applikation, ohne weitere Einschränkungen
Lesen lokaler Dateien			
Schreiben lokaler Dateien			
Zugriff auf Dateinformation			
Löschen von Dateien			
Ausführen anderer Programme			
Auslesen der Property user.name			
Zugriff auf Netzwerkport des Herkunfts-Servers des Applets			
Zugriff auf Netzwerkport eines beliebigen Servers			
Dynamisches Laden von Java-Bibliotheken			

Beenden der virtuellen Maschine mit
[exit](#)



Erzeugen von Fenstern

mit Warnhinweis



Ausführen nativen-Codes



Das Applet [openProperties](#) zeigt die Inhalte der abfragbaren Systemeigenschaften an.

CannotDisplayApplet

Beim Versuch innerhalb eines Applets eine aus Sicherheitsgründen nicht zugleassene Funktionalität auszuführen wird ein Ausnahmeereignis-Objekt der Klasse [AccessControlException](#) erzeugt.

Hinweis: Die dargestellten Einschränkungen stellen für manche Anwendungsfälle zu deutliche Restriktionen dar. Daher existiert mit den sog. *signed Applets* eine Möglichkeit einzelne Einschränkungen für bestimmte Applets gezielt zu öffnen.

Drei Komponenten wirken bei der Realisierung des Java-Sicherheitsmodells zusammen:

1. Der binäre Bytecode wird nicht direkt durch die physische Hardware ausgeführt, sondern durch die virtuelle Maschine interpretiert.
2. Der Security Manager überwacht alle sicherheitsrelevanten Operationen.
3. Applets mit weitergehenden Berechtigungen müssen sich explizit identifizieren.

Abspielen von Audiodateien:

War diese Funktionalität bis zum Erscheinen der Java 2 Plattform ausschließlich Applets vorbehalten steht sie nun auch in Java-Applikationen zur Verfügung.

Generelles Vorgehen:

Laden einer Audiodatei mittels der Methodenfamilie [getAudioClip](#) über eine gültige URL.

Im Anschluß kann das rückgegebene Objekt, welches die [AudioClip](#)-Schnittstelle implementiert abgespielt werden. Java ab Version 2 unterstützt MIDI-Dateien sowie Audiodateien der Formate RMF, WAVE, AIFF und AU.

Zur Soundausgabe in Applikationen wurde die Klassenmethode [newAudioClip\(URL\)](#) eingeführt, die eine Audiowiedergabe auch ohne Existenz eines Applet-Objekts ermöglicht.

Üblicherweise werden im Zusammenhang mit der Audiowiedergabe [Threads](#) verwendet um die Sounds mit mit anderen Ereignissen zu synchronisieren.

Anzeigen von Graphiken:

Bilder in den Formaten GIF, PNG oder JPEG können durch die Methoden [getImage](#) geladen werden.

Die Anzeige erfolgt durch [drawImage\(...\)](#).

Das GIF98a-Animationsformat wird ebenfalls unterstützt.

Anmerkung: Alternativ kann auch signaturgleiche Methode [getImage](#) der Klasse [java.awt.Toolkit](#) benutzt werden.

```
(1)import java.applet.Applet;
(2)import java.awt.Graphics;
(3)import java.awt.GridBagLayout;
(4)import java.awt.Button;
(5)import java.awt.TextField;
(6)import java.awt.Image;
(7)import java.awt.image.ImageObserver;
(8)import java.awt.event.ActionEvent;
(9)import java.awt.event.ActionListener;
(10)
(11)public class GfxViewer extends Applet implements ActionListener {
(12)    protected TextField graphicsName;
(13)    Image imgToDisplay = null;
(14)
(15)    public void init() {
(16)        graphicsName = new TextField ("enter URL of image to view", 50);
(17)        Button loadBtn = new Button ("view");
(18)        GridBagLayout gbl = new GridBagLayout();
```

```

(19)         setLayout(gbl);
(20)
(21)         loadBtn.addActionListener(this);
(22)         add(loadBtn);
(23)         add(graphicsName);
(24)     } //init
(25)
(26)     public void actionPerformed(ActionEvent event) {
(27)         imgToDisplay = getImage( getDocumentBase(), graphicsName.getText() );
(28)         repaint();
(29)     } //actionPerformed()
(30)
(31)     public boolean imageUpdate(Image img, int infoflags, int x, int y, int
width, int height) {
(32)         repaint();
(33)         if ((infoflags & ImageObserver.SOMEBITS) != 0) {
(34)             System.out.println("part just loaded:"+x+" "+y+" "+ width+"
"+height);
(35)             return true;
(36)         } //if
(37)         if ((infoflags & ImageObserver.ALLBITS) != 0) {
(38)             System.out.println("image loaded completely");
(39)             return false;
(40)         } //if
(41)         //gets here when height and width of image is available
(42)         return true;
(43)     } //imageUpdate()
(44)
(45)     public void paint(Graphics g) {
(46)         if (imgToDisplay != null) {
(47)             g.drawImage(imgToDisplay, 1, 1, this);
(48)         } //if
(49)     } //paint
(50)} //class GfxViewer

```

Beispiel 71: Ein einfacher Graphik-Viewer [GfxViewer.java](#)

[Applet ausführen](#)

Das Beispiel implementiert einen einfachen Graphik-Viewer für die beiden standardmäßig unterstützten Formate.

Es verwendet bereits interaktive Komponenten aus dem [Abstract Windows Toolkit](#) (AWT) (siehe `init`- und `actionPerformed`-Methode). Die rudimentäre Funktionalität erschöpft sich darin, eine Graphikdatei aus dem aktuellen Verzeichnis (ermittelt per `getDocumentBase()`) zu laden, und im aktuellen Fenster anzuzeigen.

Zunächst wird das Bild per `getImage`-Methode referenziert (`graphicsName.getText()` gibt den im Textfeld eingegebenen Namen als String zurück).

Der Aufruf von `repaint()` wirkt auf die gesamte Komponente. Erfolgt das Neuzeichnen, so wird die `paint`-Methode des Applets aufgerufen. Der dort platzierte Methodenaufruf `drawImage` auf dem `Graphics`-Objekt zeichnet das Bild.

Durch Überschreiben der Methode `imageUpdate` (erbt von `Component`, welches die Schnittstelle `ImageObserver` implementiert) kann der Ladevorgang des Bildes verfolgt werden. Diese Methode wird während des Ladevorganges immer wieder automatisch aufgerufen.

Anmerkungen:

- Die `getImage`-Methoden sind asynchron ausgelegt, und kehren daher sofort nach ihrem Aufruf zurück, unabhängig davon ob das referenzierte Bild existiert.
Die erforderlichen Graphikdaten werden erst beim Zeichenversuch in den Speicher geladen.
- Die `drawImage`-Methoden sind (ebenfalls) asynchron ausgelegt, und kehren sofort nach ihrem Aufruf zurück, das Laden des Bildes erfolgt ggf. in einem eigenen Thread.

Hinweis: Die Klasse [MediaTracker](#) stellt eine Implementierung zur Verfügung, welche es ermöglicht das Laden eines Bildes oder einer Audiodatei zu verfolgen.

Animationen:

...sind prinzipiell nichts anderes als die schnelle Wiedergabe verschiedener Einzelbilder. Jedoch gilt es auf einige Randbedingungen Rücksicht zu nehmen:

- Die Anzeigefrequenz der Einzelbilder sollte nicht zu niedrig gewählt sein, sonst erscheint die Animation ruckelig, allerdings auch nicht zu hoch, sonst gehen dem Auge des Betrachters Einzelbilder verloren.
- Beim Neuzeichnen der visuellen Komponente durch die `repaint()`-Methode kann es zu Flimmer- oder Flackereffekten kommen.
- Durch den Darstellungsaufwand sollte der übrige interaktive Programmablauf nicht gestört werden.

Das Beispiel zeigt die Realisierung einer simplen Uhr (Java-[Date](#)-Objekt) als animiertes Applet.

```
(1)import java.applet.Applet;
(2)import java.awt.Graphics;
(3)import java.util.Date;
(4)
(5)public class Clock extends Applet implements Runnable {
(6)    protected int counter;
(7)
(8)    public void start() {
(9)        Thread myClock = new Thread(this);
(10)        myClock.setPriority(Thread.MIN_PRIORITY);
(11)
(12)        myClock.start();
(13)    } //start()
(14)
(15)    public void run() {
(16)        while (true) {
(17)            repaint();
(18)            try {
(19)                Thread.sleep(1000);
(20)            } catch (InterruptedException ie) {
(21)                //ignore it
(22)            } //catch
(23)        } //while
(24)    } //run()
(25)
(26)    public void paint(Graphics g) {
(27)        g.drawString( new Date().toString(),50,50);
(28)    } //paint()
(29)} //class Clock
```

Beispiel 72: Eine einfache Uhr [Clock.java](#)

Applet ausführen

Der notwendige Bildschirmneuaufbau wird in einen separaten niedrig priorisierten Thread verlagert. *Hinweis:* Der neue Thread wird nicht, wie bisher per Ableitung von Thread definiert und als spezialisiertes Objekt instanziiert. Das Applet implementiert die Runnable-Schnittstelle; daher kann ein Objekt vom Typ des Applets einer Variable des Typs Thread zugewiesen werden. Erst hierdurch werden die Thread-spezifischen Methoden, wie Prioritätsänderungen, verfügbar.

Die häufigst anzutreffende Realisierung graphischer Animationen ist die Einblendung sich leicht unterscheidender Bilder, wodurch ab einer gewissen Bildfrequenz der Eindruck kontinuierlicher Bewegung entsteht (Trickfilmeffekt).

Diese Animationsvariante kann leicht mit den bisher vorgestellten Möglichkeiten realisiert werden. Hierzu werden zunächst die benötigten Bilder vollständig geladen (z.B. [getImage\(String\)](#) oder [getImage\(URL\)](#) aus der [Toolkit](#)-Klasse) und anschließend in schneller Folge am Bildschirm angezeigt (z.B. mit Methoden der [drawImage\(...\)](#)-Familie).

Im Kern ist das Verfahren identisch zur [Anzeige statischer Graphiken](#).

```
(1)import java.awt.Graphics;
(2)import java.awt.Image;
(3)import java.applet.Applet;
(4)
(5)public class AnalogousClock extends Applet implements Runnable {
(6)    static Image images[];
(7)    static int counter=0;
(8)    Thread animator;
(9)
(10)    public void init() {
(11)        images = new Image[12];
(12)        for (int i=0;i<images.length;i++)
```



```

(13)         images[i] = getImage( getDocumentBase(), "clock-"+i+".gif");
(14)     } //init()
(15)
(16)     public void start() {
(17)         while ( images[0].getWidth(this) == -1 || images[0].getHeight(this) == -1)
(18)             ;
(19)         this.resize( images[0].getWidth(this) , images[0].getHeight(this) );
(20)         animator = new Thread (this);
(21)         animator.setPriority(Thread.MIN_PRIORITY);
(22)         animator.start ();
(23)     } //start()
(24)
(25)     public void stop() {
(26)         animator = null;
(27)     } //stop()
(28)
(29)     public void run () {
(30)         while (true) {
(31)             try {
(32)                 Thread.sleep(200);
(33)             } //try
(34)             catch (InterruptedException ie) {
(35)                 //ignore it
(36)             } //catch
(37)
(38)             repaint();
(39)             if (++counter==images.length)
(40)                 counter=0;
(41)             } //while
(42)     } //run()
(43)
(44)     public void paint (Graphics g) {
(45)         System.out.println("drawing image: clock-"+counter+".gif");
(46)         g.drawImage (images[counter], 0, 0, this);
(47)     } //paint()
(48)} //class AnalogousClock

```

Beispiel 73: Analoge Uhr als Bildfolge [AnalogousClock.java](#)

[Applet ausführen](#)

Das Beispielprogramm zeichnet in einem eigenen Thread alle 200 Millisekunden den Bildschirm mit einem neuen Bild neu.

Zunächst werden durch die `init`-Methode die benötigten zwölf Graphiken in einen Image-Array geladen. Zum Ausführungsbeginn (`start`-Methode) wird zunächst die Fenstergröße des Applets (bei Ausführung im Applet-Viewer) auf die Dimensionen der ersten geladenen Graphik -- im Beispiel besitzen alle Graphiken dieselben Ausmaße, daher ist die Auswahl der Referenzgraphik unbedeutend -- gesetzt. Aufgrund der asynchronen Realisierung von `getImage()` muß auf den Abschluß des Ladevorganges gewartet werden; andernfalls liefern die Methoden `getHeight` und `getWidth` mit -1 einen ungültigen Wert zurück.

Der niederpriore Thread `animator` sorgt alle 200 Millisekunden für den notwendigen Bildschirmneuaufbau -- durch Aufruf der `repaint`-Methode --, und schaltet die Graphiken fort, wodurch der Animationseffekt entsteht.

Bei dieser Anwendung fällt ein starker **Flimmereffekt** ins Auge. Dieses (unschöne) Verhalten ist oftmals bei graphischen Java-Applets und -Applikationen anzutreffen. Zur Behebung existieren einige Standardlösungsvarianten:

Die einfachste Lösung liegt im Überschreiben der von `Component` ererbten Methode `update()`. Im Standardfalle löscht `update()` den gesamten Bildschirmbereich und ruft im Anschluß `paint()` zum vollständigen Neuzeichnen auf.

Im vorliegenden Beispiel ist jedoch das dem Neuzeichnen vorausgehende Löschen nicht notwendig, da die nachfolgende Graphik ohnehin die vorhergehende vollständig überdeckt. Daher kann die `update`-Methode auf den reinen Aufruf von `paint` beschränkt werden. Das zusätzliche Codefragment ergibt sich als:

```

public void update(Graphics g) {
    paint(g);
}

```



```
} //update()
```

[ergänzter Quellcode](#)
[ergänztes Applet ausführen](#)

Diese einfache Maßnahme hilft oftmals bei auftretenden Problemen, jedoch ist sie nur für den eng umrissenen Problemkreis anwendbar, in dem neue graphische Elemente die vorhergehenden vollständig überdecken. Andernfalls kann es zu Darstellungsfehlern kommen. Daher kommt bei realen Problemstellungen zumeist **double Buffering** zum Einsatz.

Der Mechanismus des double bufferings macht es sich zu Nutze, daß auch im nicht sichtbaren Bereich des Bildschirms graphische Operationen vorgenommen werden können. Die Anwendung verläuft üblicherweise in zwei Schritten: zunächst wird der darzustellende Bildschirminhalt verdeckt vorbereitet, und im zweiten Schritt in einer einzigen -- im Allgemeinen sehr performant ausgeführten -- Kopieroperation in den sichtbaren Bereich übertragen.

Ablaufprinzip:

1. Erzeugung eines [Image](#)-Objekts beliebiger Größe (gesteuert durch die beiden Übergabeparameter) mittels [createImage\(int, int\)](#).
`createImage(int, int)` ist in der Klasse [Component](#) definiert, und wird an Applet vererbt.
2. Erzeugung des Zeichenbereichs (*graphics context*) durch die Methode [getGraphics\(\)](#) auf dem Image-Objekt.
3. Zeichenoperationen auf dem Graphikkontext.
4. Anzeigen des Graphikkontextes innerhalb der `paint`-Methode mit einer Methode aus der [drawImage\(...\)](#)-Familie.

Codeschablone:

```
Image im = createImage(100,100);
Graphics graph = im.getGraphics();
//Operationen auf graph

public void paint(Graphics g) {
    g.drawImage(im, ...);
} //paint()
```

In `paint()` werden die vorgezeichneten Image-Objekte ohne weitere Veränderungen angezeigt. Je nach Komplexität der Anwendung können so viele Image-Objekte wie benötigt parallel gehalten und bei Bedarf angezeigt werden.

3.2.4 Collection API

Die Collection API liefert als Bestandteil der Java Standard API Hilfsklassen zum Umgang mit generischen Objektsammlungen.

Der Begriff *Collection* bezeichnet Datenstrukturen wie Listen, Stacks, Schlangen, Bäume und Mengen im mathematischen Sinne.

Das **Collection Framework** der Java-API wird aus den vorgegebenen Schnittstellen, den Operationen und ihrer definierten Semantik, den konkreten Implementierungen der wiederverwendbaren Datenstrukturen und der zugrundeliegenden Algorithmen gebildet. Das wohl bekannteste Beispiel eines solchen Rahmens dürfte die [C++ Standard Template Library](#) (Abk. STL) und die Collection-Klassen der Programmiersprache SmallTalk sein. Neben dem Collection Framework der Java API existiert mit der [Generic Collection Library for Java](#) (Abk. JGL) auch noch eine frei verfügbare Adaption der STL an Java.

Da Java bis zur Version 1.5 keine parametrische Polymorphie unterstützte sind diese Klassen auch (noch) in ihrer generischen Fassung umgesetzt. Ihr Einsatz unter Nutzung der *Java Generics* wird Gegenstand des [nächsten Kapitels](#) sein.

Um ihre Einsetzbarkeit möglichst universell zu gestalten erlauben alle Collection-Klassen die Verwaltung von Objekten des Typs `Object`, dem gemeinsamen Basistyp aller Java-Objekte.

Durch diese Mimik geht die **Typsicherheit verloren!**

Das bedeutet, daß in jeder Collection ausschließlich Objekte des Typs `Object` verwaltet werden können, die vor dem Einstellen in die Sammlung (implizit und automatisch) in diesen Typ konvertiert werden. Diese Typumwandlung ist als *up-cast* typsicher möglich, da alle Java-Klassen

immer direkte oder transitive Subklasse von `Object` sind. Bei der Entnahme von `Collection`-Elementen ist jedoch die Rekonstruktion des Ursprungstypen notwendig. Diese Typumwandlung ist als *down-cast* explizit anzugeben. Beim Konvertierungsversuch in den falschen Typ kann es hier zu einer `ClassCastException` kommen.

Unter praktischen Gesichtspunkten bietet sich die Vorschaltung einer Typprüfung durch `instanceof` an.

Andererseits schafft die Abkunft aller Java-Objekte von `Object` auch erst die Voraussetzungen für ein generisches `Collection` Framework zur Verwaltung beliebiger Java-Objekte.

Hiermit sind im Speziellen drei Methoden der Klasse `Object` gemeint:

- `boolean equals(Object)`

Erlaubt den Vergleich beliebiger Java-Objekte auf Identität. Daher liefert sie auch bei der Gleichheit aller Attributwerte `false` zurück, da sich die Objekte in ihrer (für den Benutzer nicht sicht- und zugreifbare) Objektidentität unterscheiden.

`equals()` bildet eine Äquivalenzrelation auf `Object` und ist daher reflexiv, symmetrisch und transitiv.

Insondere duplikatfreie Sammlungen greifen auf diese Methode zurück, um zu ermitteln, ob sich ein aufzunehmendes Objekt bereits in der Sammlung befindet.

- `hashCode()`

Die native Methode liefert für einen über die gesamte Laufzeit des Java-Programms unveränderten `int`-Wert als eindeutige Kennung jedes Objekts. Identische Objekte (d.h. Objekte für die `equals` den Wert `true` liefert) haben identische Kennungen.

Die über Hashfunktion jedem Objekt eindeutig zuordenbare `int`-Repräsentation kann als Hash-Code verwendet werden.

Bereits seit Version 1.0 des JDKs stehen die Klassen `Bitset`, `Dictionary`, `Hashtable`, `Stack` und `Vector` zur Verfügung.

Diese wurde mit dem JDK v1.2 um einige weitere Klassen und Schnittstellen ergänzt.

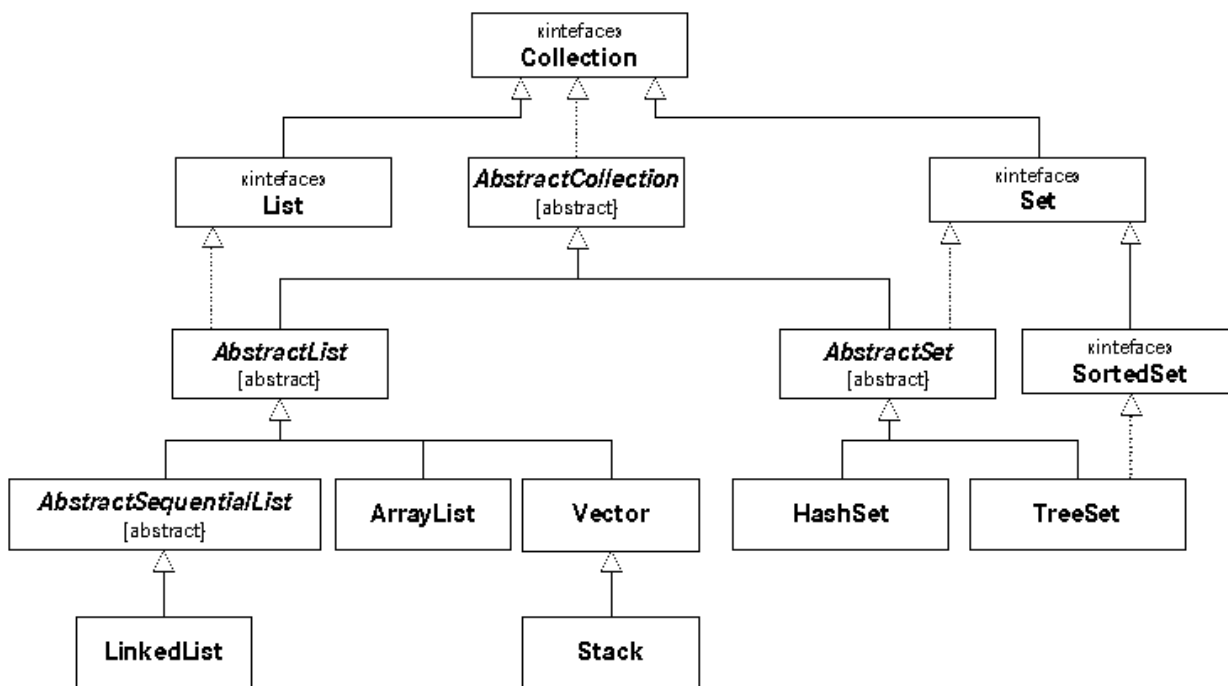


Abbildung 56: Die Collection-Klassen und ihre Schnittstellen (click on image to enlarge!)

Mit Ausnahme von `Vector` und `Stack` wurden alle dargestellten Klassen und Schnittstellen mit der Java-2-Plattform eingeführt.

Grundlegende Konzepte:

Gemeinsam sind allen `Collection`-Varianten die basalen Operationen

- Erzeugen der Sammlung
- Einstellen von Elementen
- Entnehmen von Elementen
- Zugriff auf einzelne Elemente
- Löschen der Sammlung

Darüberhinaus bieten die verschiedenen `Collection`-Klassen auch Möglichkeiten zur Suche, Sortierung usw. an.

Interessante (Realisierungs-)Details einer jeden Collection-Klasse sind weiter:

- interne Umsetzung
Die tatsächliche technische Realisierung einer wie auch immer präsentierten abstrakten Menge spielt eine entscheidende Rolle hinsichtlich Skalierbarkeit und damit verbundenem Zugriffsverhalten.
- unterstützte Zugriffsmodi
Sie bestimmen wesentlich die praktische Verwendbarkeit einer Sammlung in existierenden Algorithmen. Neben den gebotenen Zugriffsmodi spielt auch ihre Realisierung -- konkret die Anzahl der notwendigen Zugriffe bis das gesuchte Datum gefunden ist -- eine entscheidende Rolle.
- Duplikatfreiheit und Anordnungseigenschaft
Der Sammlungsgedanke an sich unterstellt zunächst weder eine (ungeordnete duplikatfreie) Menge im mathematischen Sinne, noch eine (nach welchen Kriterien auch immer) angeordnete Aufzählung der Elemente.

Das Standard Collection Framework umfaßt folgende ausprogrammierte Klassen:

Klasse	Charakteristika / Einsatzgebiete
ArrayList	Dynamischer Array, der intern durch statischen (Java nativen) Array realisiert ist. Lesender und schreibender indizierter Zugriff auf beliebige Elementposition. Ähneln Vector, ist jedoch nicht threadsicher.
Vector	Bereits im JDK v1.0 enthalten, mittlerweile um Schnittstelle List angereichert. Threadsichere Variante der ArrayList.
LinkedList	Implementierung einer doppelt verketteten Liste. Dadurch konstanter Aufwand beim Einfügen und Löschen an jeder beliebigen Listenposition, jedoch höherer Aufwand beim Anfügen und Entnehmen von letzter Listenposition als bei ArrayList.
Stack	Stack nach dem <i>last-in-first-out</i> -Prinzip. Wie Vector bereits seit dem JDK v1.0 angeboten, und daher nicht originärer Bestandteil des Collection Frameworks. Ebenfalls wie Vector threadsicher realisiert.
Hashtable	Hash-Tabelle die Einstellung von Werten über einen eindeutigen Schlüssel. Bereits mit JDK v1.0 eingeführt; nicht Bestandteil des Collection Frameworks.
HashSet	Realisierung einer Menge im mathematischen Sinne, d.h. anordnungslose (assoziative) Speicherung paarweise verschiedener (Duplikatfreiheit!) Elemente.
TreeSet	Gleiches Prinzip wie HashSet, jedoch interne Speicherorganisation als Baum. Die Elemente werden, sofern nicht anders angegeben, in ihrer <i>natürlichen</i> Ordnung abgelegt. Hierzu muß durch die Klasse jedes abzulegenden Objektes die Comparable -Schnittstelle umgesetzt sein, d.h. die <code>compareTo</code> -Methode implementiert werden.
HashMap	Erweiterung des HashSets um Schlüssel. Ablage von beliebigen Werten (auch Duplikaten) gemeinsam mit einem eindeutigen Schlüssel. (Dieser Strukturtyp wird auch als <i>Bag</i> bezeichnet.) Die Einträge werden intern durch eine Hash-Tabelle verwaltet. Äquivalent zur Hashtable, abgesehen von der mangelnden Threadsicherheit und den zugelassenen Null-Elementen.
TreeMap	Erweiterung der HashMap auf eine Baum-basierte (rot-schwarz-Baum) Ablage eines Wertes (Object) mit einem eindeutigen Schlüssel.
WeakHashMap	Prinzipiell identisch zur HashMap, jedoch werden Schlüssel-Wert-Paare entfernt, sobald die letzte externe Referenz auf den Schlüssel ungültig wird.

Ein einführendes Beispiel:

Die Klasse [Vector](#) ist dem „klassischen“, d.h. bereits mit JDK v1.0 eingeführten, Anteil der Collection-API zuzurechnen. Sie stellt im Grunde eine dynamische Arrayimplementierung zur Verfügung, die intern als statischer Array realisiert ist. Wie bei Arrays üblich erfolgt der Element-Zugriff durch einen ganzzahligen `int`-Index.

Durch den Ausbau der Collection-API in der Java-2-Plattform wurde die Vector-Implementierung nochmals überarbeitet und an die heute Struktur (namentlich die neu eingeführte Schnittstelle [List](#)) adaptiert.

```

(1)import java.io.FileDescriptor;
(2)import java.io.FileNotFoundException;
(3)import java.io.FileReader;
(4)import java.io.IOException;
(5)import java.io.LineNumberReader;
(6)import java.util.Vector;
(7)
(8)public class VectorEx1 {
(9)    public static void main(String[] args) {
(10)        LineNumberReader lnr;
(11)        String line;
(12)        Vector vec = new Vector();
(13)
(14)        try {
(15)            lnr = new LineNumberReader(new FileReader( FileDescriptor.
in ) );
(16)
(17)            while ( (line = lnr.readLine()).compareTo("exit")!=0 ) {
(18)                vec.add(line);
(19)            } //while
(20)            lnr.close();
(21)
(22)            for (int i=0; i<vec.size(); i++)
(23)                System.out.println("element no. "+i+": "+vec.get
(i) );
(24)        } catch (FileNotFoundException fnfe) {
(25)            System.out.println("cannot find stdin!");
(26)        } catch (IOException ioe) {
(27)            System.out.println("an IOException occurred\n"+ioe.toString
()+"\n"+ioe.getMessage() );
(28)        } //catch
(29)    } //main()
(30)} //class VectorEx1

```

Beispiel 74: Vektor-Operationen [VectorEx1.java](#)

Das Programm liest solange von der Standardeingabe zeilenweise Zeichenketten ein, bis der String *exit* eingegeben wird. Jede eingelesene Zeichenkette wird -- mit der Methode `add()` -- als Vector-Element angelegt.

Nach Beendigung des Einlesevorganges werden die gespeicherten Vector-Elemente in der Reihenfolge der Anlage ausgegeben. Dies geschieht per indiziertem Zugriff auf jedes Element mit der Methode `get(int)`. Der Zugriff erfolgt nicht mehr, wie bei Array üblich per direkter Indizierung in eckigen Klammern, sondern jetzt über eine eigene Methode, die jedoch den gewohnten Sicherheitsmechanismus der `ArrayOutOfBoundsException` beim Zugriffsversuch außerhalb der Vector-Grenzen beibehält.

Beispielablauf:

```

$java vectorEx1
this
is
a
simple
test
containing an

empty
line
exit
element no. 0: this
element no. 1: is
element no. 2: a
element no. 3: simple
element no. 4: test
element no. 5: containing an
element no. 6:
element no. 7: empty
element no. 8: line

```

Die **Skalierbarkeit eines Vektor-Objekts** kann durch Konstruktor beeinflußt werden. Der erlaubt die Angabe zweier Freiheitsgrade, der initialen Kapazität und eines Wachstumsfaktors. Werden, wie im Beispiel, keine Parameter übergeben, so wird gegenwärtig (d.h. im JDK v1.3) ein Vector der Größe zehn mit einem Wachstumsfaktor von Null erzeugt.

Wird durch die Einfügemethode `add` die Aufnahmekapazität des Vectors überschritten, so wächst er um den als *capacityIncrement* angegebenen Faktor, oder verdoppelt seine gegenwärtige Größe falls kein Wachstumsfaktor angegeben wurde. In beiden Fällen sind jedoch alle Vector-Elemente in einen neu zu erzeugenden Array zu transferieren, wodurch es bei umfangreichen Vektoren zu Zeitverlusten kommen kann. Die in der gezeigten Implementierung benutzte Umsetzung durch `System.arraycopy` läuft als native Methode jedoch sehr performant ab. Vor grösseren Einfügevorgängen bekannten Umfangs bietet sich jedoch die explizite Allokation des notwendigen Speicherplatzes durch Aufruf der Methode [`ensureCapacity\(int\)`](#) an, die sicherstellt, daß mindestens die im Parameter übergebene Anzahl von Objekten ohne zusätzliche Allokations- und Kopiervorgänge in den Vector eingestellt werden kann.

Der nachfolgende Codeausschnitt aus den Java-Quellen läßt das interne Verhalten von Vector anschaulich werden.

```
package java.util;

public class Vector extends AbstractList implements List, Cloneable, java.io.
Serializable {
    protected Object elementData[];
    protected int capacityIncrement;
    protected int elementCount;

    public Vector() {
        this(10);
    } //constructor

    public Vector(int initialCapacity) {
        this(initialCapacity, 0);
    } //constructor

    public Vector(int initialCapacity, int capacityIncrement) {
        ...
        this.elementData = new Object[initialCapacity];
        ...
    } //constructor

    public synchronized Object get(int index) {
        if (index >= elementCount)
            throw new ArrayIndexOutOfBoundsException(index);

        return elementData[index];
    } //get()

    public synchronized boolean add(Object o) {
        ...
        ensureCapacityHelper(elementCount + 1);
        elementData[elementCount++] = o;
        return true;
    } //add()

    private void ensureCapacityHelper(int minCapacity) {
        int oldCapacity = elementData.length;
        if (minCapacity > oldCapacity) {
            Object oldData[] = elementData;
            int newCapacity = (capacityIncrement > 0) ? (oldCapacity +
capacityIncrement) : (oldCapacity * 2);
            if (newCapacity < minCapacity) {
                newCapacity = minCapacity;
            } //if
            elementData = new Object[newCapacity];
            System.arraycopy(oldData, 0, elementData, 0, elementCount);
        } //if
    } //ensureCapacityHelper()

    public Enumeration elements() {
        return new Enumeration() {
            int count = 0;

```

```

    public boolean hasMoreElements() {
        return count < elementCount;
    } //hasMoreElements()

    public Object nextElement() {
        synchronized (Vector.this) {
            if (count < elementCount) {
                return elementData[count++];
            } //if
        } //synchronized
        throw new NoSuchElementException("Vector Enumeration");
    } //nextElement()
}; //elements()
} //class Vector

```

Einen einfacheren Weg zur **Traversierung eines Vectors** bietet die **Schnittstelle [Enumeration](#)** an.

Ein solches Objekt wird durch die Methode [elements\(\)](#) zurückgegeben. Sie abstrahiert nochmals vom indizierten Zugriff, und stellt mit den Methoden [hasMoreElements\(\)](#) und [nextElement\(\)](#) generische Zugriffsprimitiven bereit, die so auch in anderen Collection-Klassen Anwendung finden. *Nebenbemerkung:* Die Implementierung von [elements\(\)](#) stellt ein (schönes) Beispiel für die Verwendung [anonymer innerer Klassen](#) dar.

Bei Modifikationen durch nebenläufig ausgeführte Threads am zugrundeliegenden Vector nachdem ein Enumeration-Objekt erzeugt wurde, kann es zu Konsistenzproblemen kommen. Daher wird seit dem JDK v1.2 die Verwendung der Schnittstelle [ListIterator](#) gegenüber der weiter angebotenen Enumeration empfohlen. Ihre Implementierung ist *fail-fast*, d.h. sie erzeugen ein Ausnahmeereignis-Objekt der Klasse [ConcurrentModificationException](#) im Falle nebenläufiger Modifikation. Zusätzlich erlaubt der Iterator selbst die Entnahme von Elementen aus der zugrundeliegenden Sammlung.

Der zu einer Collection gehörige Iterator wird über die (von List geerbte) Methode [ListIterator\(\)](#) geliefert.

Vorwärtsreferenz: [Beispiel zur Nutzung eines Iterators](#)

Mengen im mathematischen Sinne (d.h. duplikatfrei und ungeordnet) lassen sich durch die Klasse **HashSet** realisieren.

Mit den (ererbten) Methoden [addAll](#) (Vereinigung), [retainAll](#) (Durchschnitt), [removeAll](#) (Differenz) können die grundlegenden Mengenoperationen realisiert werden.

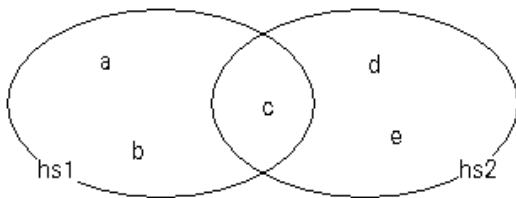


Abbildung 57: Eingabedaten des Beispiels (click on image to enlarge!)

```

(1)import java.util.HashSet;
(2)import java.util.Iterator;
(3)
(4)public class HashSetEx1 {
(5)    public static void main(String[] args) {
(6)        HashSet hs1 = new HashSet();
(7)        HashSet hs2 = new HashSet();
(8)        HashSet hs3;
(9)
(10)       hs1.add( new String("a") );
(11)       hs1.add( new String("b") );
(12)       hs1.add( new String("c") );
(13)
(14)       hs2.add( new String("c") );
(15)       hs2.add( new String("d") );
(16)       hs2.add( new String("e") );
(17)
(18)       System.out.println("hs1="+hs1.toString() );
(19)       System.out.println("hs2="+hs2.toString() );
(20)
(21)       //union
(22)       hs3 = new HashSet( hs1 );
(23)       hs3.addAll(hs2);

```

```

(24)      System.out.println("UNION(hs1,hs2)="+hs3.toString() );
(25)
(26)      hs3 = null;
(27)      //difference
(28)      hs3 = new HashSet( hs1 );
(29)      hs3.removeAll( hs2 );
(30)      System.out.println("hs1 WITHOUT hs2 =" +hs3.toString() );
(31)
(32)      hs3 = null;
(33)      //intersection
(34)      hs3 = new HashSet( hs1 );
(35)      hs3.retainAll( hs2 );
(36)      System.out.println("INTERSECTION(hs1,hs2)="+hs3.toString() );
(37)
(38)      hs3 = null;
(39)      //symmetric difference
(40)      hs3 = new HashSet( hs1 );
(41)      hs3.addAll( hs2 );
(42)      HashSet tmp1 = new HashSet( hs1 );
(43)      tmp1.retainAll( hs2 );
(44)      hs3.removeAll( tmp1 );
(45)      System.out.println("Symmtetric Difference(hs1,hs2)="+hs3.toString
() );
(46)
(47)      hs3 = null;
(48)      //cartesian product
(49)      hs3 = new HashSet();
(50)      String c1;
(51)
(52)      for( Iterator hs1It = hs1.iterator(); hs1It.hasNext(); ) {
(53)          c1 = hs1It.next().toString();
(54)          for( Iterator hs2It = hs2.iterator(); hs2It.hasNext(); )
(55)          {
(56)              hs3.add( new String( c1 + (hs2It.next()).toString
() ) );
(57)          } //for
(58)      } //while
(59)      System.out.println("Cartesian Product(hs1,hs2)="+hs3.toString() );
(60)  } //main()
(61)} //class HashSetEx1

```

Beispiel 75: Die grundlegenden Mengenoperationen [HashSetEx1.java](#)

BildschirmAusgabe:

```

hs1=[b, a, c]
hs2=[e, d, c]
UNION(hs1,hs2)=[b, a, e, d, c]
hs1 WITHOUT hs2 =[b, a]
INTERSECTION(hs1,hs2)=[c]
Symmtetric Difference(hs1,hs2)=[b, a, e, d]
Cartesian Product(hs1,hs2)=[ce, cd, cc, be, bd, bc, ae, ad, ac]

```

Das Beispiel zeigt neben Realisierungen der grundlegenden Mengenoperationen auch die Nutzung eines Iterators anstelle der veralteten Enumeration.

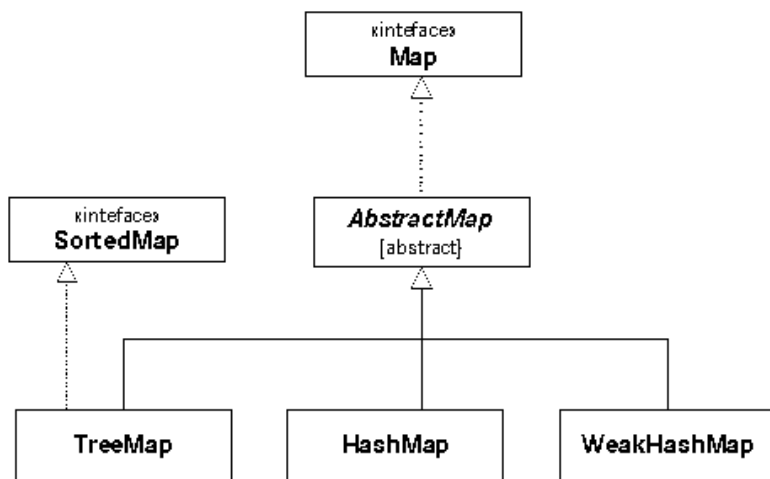


Abbildung 58: Die ... Map-Klassen (click on image to enlarge!)

Die von der abstrakten Basisklasse [AbstractMap](#) abgeleiteten konkreten Klassen [HashMap](#), [TreeMap](#) und [WeakHashMap](#) stellen eine von der Collection-Hierarchie losgelöste besondere Variante allgemeiner Object-Sammlungen zur Verfügung.

Ihnen allen gemein ist die Speicherung eines eindeutigen Zugriffsschlüssels, zusätzlich zu den Nutzinformationen der verwalteten Objekte.

Die Klasse [HashMap](#) organisiert die abgelegten Objekte intern über eine Hashfunktion, [TreeMap](#) über einen rot-schwarz-Baum.

[WeakHashMap](#) ist prinzipiell identisch zur [HashMap](#), jedoch werden Schlüssel-Wert-Paare entfernt, sobald die letzte externe Referenz auf den Schlüssel ungültig wird.

Das Beispielprogramm ermittelt die Auftrittshäufigkeit von einzelnen Worten oder Zahlen im Eingabestrom. Hierfür wird der bekannte [StreamTokenizer](#) zur entsprechenden Eingabefilterung benutzt.

Grundgedanke des realisierten Algorithmus ist es, das Wort oder die Zahl als Schlüssel in der [HashMap](#) zu nutzen, und die bisher ermittelten Vorkommen als Nutzinformation in einem Integer-Objekt abzulegen.

```

(1)import java.io.StreamTokenizer;
(2)import java.io.FileReader;
(3)import java.io.FileDescriptor;
(4)import java.io.IOException;
(5)import java.util.HashMap;
(6)import java.util.Map;
(7)
(8)public class WordCount {
(9)    private static final Integer ONE = new Integer(1);
(10)
(11)    public static void main(String[] args) {
(12)        Map m = new HashMap();
(13)
(14)        try {
(15)            StreamTokenizer st = new StreamTokenizer( new FileReader
( FileDescriptor.in ) );
(16)            int res;
(17)            Integer freq;
(18)
(19)            while( (res = st.nextToken()) == StreamTokenizer.TT_WORD ||
res == StreamTokenizer.TT_NUMBER) {
(20)                if (res == StreamTokenizer.TT_WORD)
(21)                    freq = (Integer) m.get( st.sval );
(22)            else
(23)                freq = (Integer) m.get( new String(""+st.nval) );
(24)
(25)            if (res == StreamTokenizer.TT_WORD)
(26)                m.put( st.sval, (freq==null ? ONE : new Integer(freq.intValue
() + 1)));
(27)            else
(28)                m.put( new String(""+st.nval), (freq==null ?
ONE : new Integer(freq.intValue() + 1)));
(29)        } //while
(30)        System.out.println(m.size()+" distinct words detected:");
  
```



```

(31)      System.out.println(m);
(32)      } catch (IOException ioe) {
(33)          System.out.println("an IOException occurred\n"+ioe.getMessage
() );
(34)      } //catch
(35)  } //main()
(36)} //class WordCount

```

Beispiel 76: Ermittelt die Auftrittshäufigkeit eines Wortes [WordCount.java](#)

Beispielablauf:

Die Datei [testfile.asc](#):

```

the quick brown fox jumps over the lazy dog
1 2 3 123 1 2 X

```

```

$java wordCount < testfile
13 distinct words detected:
{X=1, lazy=1, fox=1, quick=1, 1.0=2, over=1, the=2, dog=1, 123.0=1, brown=1, 2.0=2,
3.0=1, jumps=1}

```

Die Positionierung eines Elements innerhalb einer der Hash-Sammlungen (HashMap, HashSet und Hashtable) wird während der Ausführung der put-Methode durch eine Hashfunktion, die auf die Methode hashCode der Klasse Object zurückgreift, bestimmt.

Beim Erzeugungsvorgang jedes Hash-...-Objekt kann der Speicherplatzbedarf und das Laufzeitverhalten über zwei Parameter beeinflusst werden. Die *initiale Kapazität* (engl. *initial capacity*) definiert die Anzahl der Einträge in der Hashtabelle, und der *Lastfaktor* (engl. *load factor*) den maximalen Füllungsgrad einer Hashtabelle bevor automatisch eine Kapazitätserweiterung durchgeführt wird.

Werden die beiden Parameter bei der Erzeugung nicht angegeben, so wird vorgabegemäß eine Datenstruktur mit Freiraum für elf Objekte, und einem Lastfaktor von 0.75 erzeugt. Mithin wird die erste Kapazitätserweiterung beim Einfügeversuch des neuen Elements durchgeführt.

Algorithmen auf Objektsammlungen vom Typ Array

... werden durch die Klasse [Arrays](#) in Form statischer Methoden angeboten.

- [asList\(Object\[\]\)](#)
Erzeugt eine Liste fester Größe aus einem Array beliebiger Objekte.
- [binarySearch\(...\)](#)
Stellt für die verschiedenen [Primitivtypen](#)-Arrays und allgemeine Object-Arrays eine Suche nach beliebigen Inhaltswerten zur Verfügung.
- [equals\(...\)](#)
Vergleicht zwei Arrays von identischem Typ auf inhaltliche Gleichheit.
- [fill\(...\)](#)
Ersetzt alle Elemente eines Arrays durch das Übergebene.
- [sort\(...\)](#)
Sortiert übergebenen Array mit einer modifizierten Variante des Quicksort-Algorithmus.

Algorithmen auf Objektsammlungen vom Typ List

... werden durch die Klasse [Collections](#) in Form statischer Methoden angeboten.

- [binarySearch\(...\)](#)
Führt eine binäre Suche nach einem gegebenen Wert in einer Listen-artigen (d.h. Collection-Klasse welche die Schnittstelle List implementiert) durch.
Die gegenwärtige Implementierung nimmt jedoch AbstractSequentialLists und deren Subklassen (in der Standard-API realisiert: LinkedList) explizit von dieser Mimik aus, und realisiert für diese Sammlungstypen eine sequentielle Suche.
- [copy\(List, List\)](#)
Kopiert Listenelemente in eine andere Liste.
- [Enumeration\(Collection\)](#)
Liefert für jeden Collection-Typ eine Enumeration zur Traversierung zurück.
Anmerkung: Leider existiert derzeit keine Äquivalent das die modernere und sicherere Variante Iterator zurückliefern würde.
- [fill\(List, Object\)](#)
Ersetzt alle Elemente einer Liste durch das gegebene Object.
- [min\(...\)](#) und [max\(...\)](#)
Liefen das Minimum, bzw. Maximum, einer Liste, ggf. unter Anwendung einer speziellen

Vergleichsoperation. Die Suche wird nach der Methode des sequentiellen Schwellenwertes durchgeführt.

- [`nCopies\(Object\)`](#)
Liefert eine Liste mit n Kopien eines bestimmten Objekts.
- [`reverse\(List\)`](#)
Kehrt die Elementordnung für eine Liste um.
- [`reverseOrder\(\)`](#)
Liefert ein [`Comparator`](#)-Objekt zurück welches die Anordnung der Liste umkehrt.
- [`shuffle\(List\)`](#)
Permutiert Objekte der Liste.
- [`sort\(...\)`](#)
Sortiert Liste nach einem modifizierten Mergesort-Verfahren.
Die Verwendung des Mergesort-Algorithmus ist nicht zwingend, API-Implementierungen können ihn gegen durch Algorithmen ersetzen. Einzige Bedingung hierbei ist die Stabilität. D.h. die Positionserhaltung gleicher Elemente; sie sollen durch den Sortiervorgang nicht verschoben werden. In der Implementierung wird zunächst aus der Liste in ein Object-Array erzeugt, auf dem dann die sort-Methode der Klasse Arrays aufgerufen wird, die den Mergesort dann für Objects implementiert.
- [`synchronizedCollection\(Collection\)`](#), [`synchronizedList\(List\)`](#), [`synchronizedMap\(Map\)`](#), [`synchronizedSet\(Set\)`](#), [`synchronizedSortedMap\(SortedMap\)`](#) und [`synchronizedSortedSet\(SortedSet\)`](#)
Liefere eine threadsichere Variante der jeweiligen Eingabedatenstruktur zurück.
- [`unmodifiableCollection\(Collection\)`](#), [`unmodifiableList\(List\)`](#), [`unmodifiableMap\(Map\)`](#), [`unmodifiableSet\(Set\)`](#), [`unmodifiableSortedMap\(SortedMap\)`](#) und [`unmodifiableSortedSet\(SortedSet\)`](#)
Liefere eine unveränderbare Sicht auf die jeweilige Eingabedatenstruktur zurück.

Anmerkung zur Verwendung der beiden Sortierverfahren *Mergesort* und *Quicksort* in den Klassen Arrays und Collections:

Beim ersten Hinsehen verwundert die Implementierung zweier verschiedener Sortierverfahren ein wenig, nicht zuletzt wegen des Wechsels des Sortierverfahrens in der Implementierung der sort-Methoden der Klasse Arrays. Wird dort doch für alle Sortieroperationen auf Primitivtypen-Arrays eine modifizierte Quicksort-Variante eingesetzt, jedoch für Object-Arrays zu Mergesort übergegangen.

Dies liegt in der Natur der beiden Sortierverfahren begründet. Quicksort ist eines der bekanntesten und vermutlich auch am besten erforschtesten Sortierverfahren; mit guten Leistungsdaten im allgemeinen Falle. Jedoch kann der Sortiervorgang im schlechtesten Falle bis zu n^2 Schritte benötigen. Insbesondere für große n ist daher Mergesort -- mit seinen garantierten $n \log(n)$ Schritten effizienter.

Allerdings benötigt Quicksort deutlich weniger Speicherplatz (eigentlich nur den Stack durch die rekursiven Aufrufe), während Mergesort für seinen Ablauf stets den doppelten Speicherbedarf der zu sortierenden Datenstruktur für sich reklamiert.

Aus praktischen Gründen, nicht zuletzt wegen des uniformen Laufzeitverhaltens, wurde dem speicherplatzintensiveren Mergesort der Vorzug gegeben. Da in realen Anwendungen in den seltensten Fällen reine Primitivtypen-Arrays sortiert werden, nutzt der Anwender überwiegend die vorhandenen Mergesort-Implementierungen, die jedoch in (begründeten) Einzelfällen durchaus durch eigene Routinen ersetzt werden können.

3.2.5 Parametrische Polymorphie/Generics

Die wirkungsmächtigste Neuerung der Java Version 1.5 bildet die Einführung der sog. *Generics*, welche das Feld der Template-basierten Metaprogrammierung und mithin die parametrische Polymorphie eröffnen.

Das sinnvollste Einsatzfeld dieses Mechanismus bildet die Anwendung auf die Vorhandenen Klassen der Collection-API zur Realisierung typsicherer Objektsammlungen.

Die allgemeine Syntax der Generics erfordert die Nachstellung, des durch spitze Winkelklammern eingeschlossenen Typnamen, nach dem Namen der so typisierten Sammlung. So wird eine Liste, die ausschließlich Objekte des Typs String enthält als `List<String>` definiert.

Als Resultat einer so definierten Sammlungsklasse wird bereits zum Übersetzungszeitpunkt die Konformität abgeprüft. Daher wird bei jedem Versuch Daten in eine solch konkretisierte Sammlung einzufügen geprüft, ob diese unter Beachtung der Typrestriktion kompatibel zum in der Deklaration angegebenen Inhaltstyp sind.

Konventionsgemäß erlaubt Java ausschließlich die Festlegung von Klassen als Inhaltstypen von Objektsammlungen; Primitivtypen sind von der Verwendung ausgeschlossen.

Das nachfolgende Beispiel zeigt die Definition einer Sammlung des Typs `LinkedList`, deren Inhaltselemente auf Ausprägungen des Typs `Integer` beschränkt sind.

```
(1)import java.util.LinkedList;
(2)
(3)public class GenTest1 {
(4)    public static void main(String[] args) {
(5)        LinkedList<Integer> integerList = new LinkedList<Integer>();
(6)
(7)        integerList.add(new Integer(42));
(8)        integerList.add( 84 );
(9)
(10)       System.out.println( integerList );
(11)    } //main()
(12)} //class GenTest1
```

Beispiel 77: Typisierung einer generischen Liste [GenTest1.java](#)

Das Beispiel zeigt die Definition einer typisierten `LinkedList`, für die bereits zum Übersetzungszeitpunkt geprüft wird, daß ausschließlich Ausprägungen der Klasse `Integer` der Liste hinzugefügt werden.

Diese Restriktion verhindert jedoch, vermöge der Nutzung des [dynamischen Boxings](#) nicht, daß Ausprägungen des Primitivtyps `int` direkt übergeben werden, da diese für den Programmierer transparent in Objekte des Type `Integer` umgewandelt werden.

Versuche Objekte oder Werte anderen Typs in die Liste aufzunehmen werden bereits zum Übersetzungszeitpunkt erkannt und als Fehler gemeldet.

Die Einführung typisierter Sammlungen bedingt auch die entsprechende Adaption der Hilfsklassen zur Traversierung. vor diesem Hintergrund zeigt das nachfolgende Beispiel die Typisierung eines Iterators. Als Konsequenz dieser Konkretisierung des Iterators liefert der Aufruf der Methode [next](#) statt der generischen `Object`-Instanz eine Ausprägung des Parametertyps `String`.

```
(1)import java.util.LinkedList;
(2)import java.util.Iterator;
(3)
(4)public class GenTest2 {
(5)    public static void main(String[] args) {
(6)        LinkedList<String> nameList = new LinkedList<String>();
(7)
(8)        nameList.add( "Berta" );
(9)        nameList.add( "Anton" );
(10)       nameList.add( "Cesar" );
(11)
(12)       Iterator<String> listIterator = nameList.iterator();
(13)       String value;
(14)       while(listIterator.hasNext()) {
(15)           value = listIterator.next();
(16)           System.out.println( value );
(17)       } //while
(18)    } //main()
(19)} //class GenTest2
```

Beispiel 78: Typisierung eines Iterators [GenTest2.java](#)

Neben der Anwendung auf bestehende Klassen des Collection-Frameworks können Generics auch für eigenerstellte Klassen eingesetzt werden.

Das Beispiel zeigt die Definition der Klasse `TopTen`, welche eine Liste von Einträgen beliebigen Typs zusammen mit einer Platzierung gestattet. Der Inhaltstyp wird hierbei als Polymorphieparameter zum Konstruktionszeitpunkt der `TopTen`-Objekte festgelegt. Innerhalb der Klasse wird zur Verwaltung der Daten eine `TreeMap` eingesetzt, die entsprechend der Initialisierung des `TopTen`-Objekts erzeugt wird um auch auf dieser Ebene die Typkonformität zu gewährleisten.

```

(1)import java.util.TreeMap;
(2)import java.util.Collection;
(3)import java.util.Iterator;
(4)
(5)public class GenTest3 {
(6)    public static void main(String[] args) {
(7)        TopTen<String> music = new TopTen<String>();
(8)        music.addAtPosition(1, "St. Anger/Metallica");
(9)        music.addAtPosition(3, "20 Jahre - Nena feat. Nena/Nena");
(10)       music.addAtPosition(2, "9/Eros Ramazotti");
(11)
(12)       music.print();
(13)    } //main()
(14)} //class GenTest3
(15)
(16)class TopTen<Type> {
(17)    private TreeMap<Integer,Type> content;
(18)    public TopTen() {
(19)        content = new TreeMap<Integer,Type>();
(20)    } //constructor
(21)    public boolean addAtPosition(int pos, Type value) {
(22)        if (content.size() == 10) {
(23)            return false;
(24)        } else {
(25)            content.put(new Integer(pos), value);
(26)            return true;
(27)        } //else
(28)    } //addAtPosition()
(29)    public Type getFromPosition(int pos) {
(30)        return content.get(new Integer(pos));
(31)    } //getFromPosition()
(32)    public void print() {
(33)        Iterator<Type> i = content.values().iterator();
(34)        while (i.hasNext()) {
(35)            System.out.println( i.next() );
(36)        } //while()
(37)    } //print()
(38)} //class TopTen

```

Beispiel 79: Anwendung von Generics für eigene Klassen [GenTest3.java](#)

Neben der Angabe von Klassen zur Typisierung einer generischen Aufzählung können, mit derselben Mächtigkeit, auch Aufzählungstypen angegeben werden. Das nachfolgende Beispiel zeigt eine Anwendung:

```

(1)import java.util.LinkedList;
(2)import java.util.HashMap;
(3)
(4)public class EnumTest4 {
(5)public static void main(String argv[]) {
(6)    enum DayOfWeek {monday, tuesday, wednesday, thursday, friday,
(7)        saturday, sunday};
(8)
(9)    LinkedList<DayOfWeek> allDays = new LinkedList<DayOfWeek>();
(10)    allDays.add(DayOfWeek.monday);
(11)    allDays.add(DayOfWeek.tuesday);
(12)    allDays.add(DayOfWeek.wednesday);
(13)    allDays.add(DayOfWeek.thursday);
(14)    allDays.add(DayOfWeek.friday);
(15)    allDays.add(DayOfWeek.saturday);
(16)    allDays.add(DayOfWeek.sunday);
(17)
(18)    HashMap<DayOfWeek,Double> sales = new HashMap<DayOfWeek,Double>();
(19)
(20)    sales.put(DayOfWeek.monday, 3000.0 );
(21)    sales.put(DayOfWeek.tuesday, 5000d );
(22)    sales.put(DayOfWeek.thursday, 7500D );
(23)    sales.put(DayOfWeek.wednesday, (double) 8000 );
(24)    sales.put(DayOfWeek.friday, 55000.0 );
(25)    sales.put(DayOfWeek.saturday, new Double("10000"));

```

```

(26)    sales.put(DayOfWeek.sunday, (new Double("0")).doubleValue() );
(27)
(28)    for (DayOfWeek w : allDays)
(29)        System.out.println(w+" "+sales.get(w));
(30)    } //main()
(31)} //class EnumTest4

```

Beispiel 80: Aufzählungstypen als Polymorphieparameter [EnumTest4.java](#)

Zwar vermeidet der Einsatz der Generics die Anwendung unsicherer expliziter Typkonversionen fast völlig, jedoch ist ihre Notwendigkeit in Einzelfällen nach wie vor gegeben.

Das nachfolgende Beispiel zeigt einen solchen Fall. Es definiert eine als Vector angelegte Sammlung von Objekten des Typs Person. Gemäß der Substitutionsregeln können auch Spezialisierungen der Person, mithin Ausprägungen der davon abgeleiteten Klasse Mitarbeiter, typkonform der Sammlung hinzugefügt werden.

Da die Methode get im allgemeinen Falle Ausprägungen der Klasse Object und unter Einsatz der Generics Ausprägungen des Parametertyps (in diesem Falle: Person) liefert würde die abgespeicherte Mitarbeiter-Instanz lediglich als Ausprägung der Klasse Person aus der Sammlung entnommen werden. Um eine korrekte Weiterverarbeitung (z.B. in Form des Zugriffs auf das Attribut personalnummer) als Mitarbeiter zu gewährleisten ist daher eine explizite Typkonversion notwendig.

```

(1)import java.util.Vector;
(2)
(3)public class GenTest4 {
(4)    public static void main(String[] args) {
(5)        Vector<Person> persL = new Vector<Person>();
(6)
(7)        Person p = new Person();
(8)        p.name = "Max Mustermann";
(9)        persL.add(p);
(10)
(11)        Mitarbeiter m = new Mitarbeiter();
(12)        m.name = "John Doe";
(13)        m.personalnummer = "1234";
(14)        persL.add(m);
(15)
(16)        Person p2 = persL.get(0);
(17)        System.out.println( p2.name );
(18)
(19)        Mitarbeiter m2 = (Mitarbeiter) persL.get(1);
(20)        System.out.println( m2.name );
(21)        System.out.println( m2.personalnummer );
(22)    } //main()
(23)} //class GenTest4
(24)
(25)class Person {
(26)    public String name;
(27)} //class Person
(28)
(29)class Mitarbeiter extends Person {
(30)    public String personalnummer;
(31)} //class Mitarbeiter

```

Beispiel 81: Explizite Typwandlung trotz Generics [GenTest4.java](#)

▲ 3.2.6 Reflection API

Mit der JDK-Version 1.1 wurde die Möglichkeit geschaffen zur Laufzeit Informationen über die verwendeten Informationsstrukturen zu gewinnen. Genaugenommen handelt es sich hier um Information über Information, die üblicherweise als *Metainformation* bezeichnet wird.

Bereits seit der Version 1.0 existierte mit der Klasse [Class](#) ein Mechanismus, der Möglichkeit zur Ermittlung des Typs im Stile der aus C++ bekannten *Runtime Type Identification* bietet.

Der **Begriff Laufzeittyp** meint den konkreten Typen eines Attributes oder einer Variable (im

Allgemeinen: einer referenzierten typisierten Speicherposition). Dieser Typ muß nicht zwingend über die gesamte Ausführungszeit mit dem zum Definitionszeitpunkt festgelegten identisch sein. Er kann sich durch typkonforme Neuzuweisung ändern.

Die Klasse Class

Zur Motivation: [Bisher](#) war es uns mit dem instanceof-Operator nur möglich zu entscheiden ob ein gegebenes Objekt Ausprägung einer bestimmten Klasse ist. Hierfür mußte der Namen der Klasse zum Ausführungszeitpunkt des Operators bekannt sein.

Die Methode [getClass\(\)](#) welche, da von Object geerbt, auf allen Java-Objekten zur Verfügung steht behebt dieses Manko.

Sie liefert ein Objekt der Klasse [Class](#) zurück welches -- neben zahlreichen weiteren Informationen -- den Namen der Klasse zur Laufzeit verfügbar werden läßt.

Class ist im Paket java.lang angesiedelt, und steht daher ohne explizite import-Anweisung in allen Java-Programmen zur Verfügung. Zu jeder Klasse im System existiert je ein solches Class-Objekt.

```
(1)public class RefEx1 {
(2)    public static void main(String[] args) {
(3)        Person p1 = new Person();
(4)        Mann m1 = new Mann();
(5)
(6)        System.out.println("runtime class of variable p1="+ p1.getClass().
getName() );
(7)        System.out.println("runtime class of variable m1="+ m1.getClass().
getName() );
(8)
(9)        p1 = m1; //up-cast is typesafe
(10)       System.out.println("runtime class of variable p1="+ p1.getClass().
getName() );
(11)    } //main()
(12)} //class RefEx1
(13)
(14)class Person {
(15)} //class Person
(16)
(17)class Mann extends Person {
(18)} //class Mann
(19)
(20)class Frau extends Person {
(21)} //class Frau
```

Beispiel 82: Ermittlung der Laufzeitklasse [RefEx1.java](#)

BildschirmAusgabe:

```
$java RefEx1
runtime class of variable p1=Person
runtime class of variable m1=Mann
runtime class of variable p1=Mann
```

Die Anwendung definiert neben der Hauptklasse drei weitere Klassen Person, Mann und Frau. Sie sind in der Weise organisiert, daß Mann und Frau als Subklassen von Person realisiert sind. Zunächst werden Variablen vom Typ Person und Mann definiert und mit Ausprägungen der Definitionstypen initialisiert. Durch den Methodenaufruf getName() auf dem durch getClass() zurückgelieferten Class-Objekt kann der Name der Laufzeitklasse im Klartext ermittelt werden. So liefert die Variable p1 welche auf eine Person-Ausprägung verweist erwartungsgemäß den Namen dieser Klasse zurück; für die Variable m1 und die Klasse Mann gilt dies analog. Durch der Zuweisung des Mann-Objekts m1 an die Variable p1 (vom Typ Person) verändert sich deren Laufzeittyp zu Mann.

Im vorangegangenen Beispiel wird ein Objekt zur Ermittlung der zugehörigen Laufzeitklasse benutzt. Ist der Klassenname bekannt, so kann durch den Methodenaufruf [forName\(String\)](#) das zur Klasse mit dem übergebenen Namen gehörige Class-Objekt ermittelt werden.

```

(1)public class RefEx2 {
(2)    public static void main(String[] args) {
(3)        try {
(4)            Class c1 = Class.forName("Person");
(5)            System.out.println("Name of class:"+c1.getName() );
(6)        } catch (ClassNotFoundException cnfe) {
(7)            System.out.println("cannot find class\n"+cnfe.toString()
+"\\n"+cnfe.getMessage() );
(8)        } //catch
(9)    } //main()
(10)} //class RefEx2
(11)
(12)class Person {
(13)} //class Person

```

Beispiel 83: Ermittlung eines Class-Objekts über den Klassennamen [RefEx2.java](#)

Bildschirmausgabe:

```

$java RefEx2
Name of class:Person

```

Die Klasse Person wird geladen, und per `getName()` ihr Name -- erwartungsgemäß Person -- ausgegeben.

Kann die Klasse nicht gefunden werden, so wird ein [ClassNotFoundException](#) Ausnahmeereignis-Objekt erzeugt.

Um unnötigen Tippaufwand zu sparen existiert mit `T.class` eine abkürzende Schreibweise für alle Javatypen `T` aus dem [Typsystem](#).

Beispiel:

```

Class c1 = Person.class;
Class c2 = int.class;
Class c3 = Double[].class;

```

Hinweis: Aus historischen Gründen liefert die Methode `getName()` für alle Array-Typen eine etwas unschöne abkürzende Syntax, die sich am [im Class-File verwendeten Format](#) orientiert:

Zunächst das Präfix `[` für jede Array-Schachtelung, gefolgt von ...

B für byte

C für char

D für double

F für float

I für int

J für long

L *classname*; für eine Klasse oder Schnittstelle

S für short

Z für boolean.

so liefert `(new int[3][4][5][6][7][8][9]).getClass().getName()` als Namen die Zeichenkette `[[[[[[[I` zurück.

Die **Erzeugung neuer Objekte aus einem existierenden Class-Objekt** erfolgt durch die Methode [newInstance\(\)](#). Sie ruft intern den parameterlosen Vorgabekonstruktor der durch das Class-Objekt vertretenen Klasse auf.

Die `newInstance`-Methode unterstützt keine Aufrufe parametrisierter Konstruktoren. Für diesen Anwendungsfall sollte auf die Klasse `Constructor` der (moderneren) `Reflection` API zurückgegriffen werden.


```

(1)public class RefEx3 {
(2)    public static void main(String[] args) {
(3)        Person p1 = new Person(); //normal object creation
(4)        p1.sayHello();
(5)
(6)        try {
(7)            Person p2 = (Person) Class.forName("Person").newInstance();
(8)            p2.sayHello();
(9)        } catch (ClassNotFoundException cnfe) {
(10)            System.out.println("cannot find class\n"+cnfe.toString()
+"\\n"+cnfe.getMessage() );
(11)        } catch (InstantiationException ie) {
(12)            System.out.println("an InstantiationException occurred\n"+ie.
toString()+"\\n"+ie.getMessage() );
(13)        } catch (IllegalAccessException iae) {
(14)            System.out.println("an IllegalAccessException occurred\n"+iae.
toString()+"\\n"+iae.getMessage() );
(15)        } //catch
(16)    } //main()
(17)} //class RefEx3
(18)
(19)class Person {
(20)    public void sayHello() {
(21)        System.out.println("hello from person!");
(22)    } //sayHello()
(23)} //class Person

```

Beispiel 84: Erzeugung eines neuen Objekts mit der Methode `newInstance` [RefEx.java](#)

Das Programm gibt zweimal die Zeichenkette *hello from person!* am Bildschirm aus. Die Erzeugung des zweiten Personen-Objektes (p2) erfolgt durch die Kopplung der Methoden `forName` und `newInstance`. Letztere liefert eine Ausprägung der Klasse `Object` zurück, weshalb -- um die Methode `sayHello` der davon abgeleiteten Klasse `Person` aufrufen zu können -- die explizite Typumwandlung erfolgen muß.

Ferner stellt die Klasse `Class` weitere Methoden zur Untersuchung der Charakteristika definierter Klassen bereit:

- [`getSuperclass\(\)`](#)
Liefert die Superklasse einer gegebenen Klasse. Für die Klasse `java.lang.Object`, Schnittstellen, alle Primitivtypen und `void` wird null zurückgeliefert.
- [`getInterfaces\(\)`](#)
Liefert die durch eine Klasse direkt implementierten Schnittstellen; ererbte Schnittstellen werden nicht geliefert.
- [`isInterface\(\)`](#)
`true` falls das `Class`-Objekt eine Schnittstelle ist, andernfalls `false`.
- [`isInstance\(Object\)`](#)
Entspricht dem `instanceof`-Operator.
- [`isAssignableFrom\(Class\)`](#)
Prüft ob die Zuweisung des als Parameter übergebenen Klassentypen an das Objekt auf dem diese Methode aufgerufen wird möglich ist.
Die Methode bietet damit direkten Zugriff auf den JVM-Opcode [`checkcast`](#); ohne jedoch bei Unmöglichkeit der Zuweisung ein Ausnahmeereignis-Objekt zu generieren.

Das Reflection-Paket

Wie bereits angedeutet wurde die Klasse `Class` mit Einführung der Reflection API in der Java Version 1.1 stark erweitert. Sie schlägt -- über die Rückgabetypen der neu geschaffenen Methoden -- eine Brücke zu den Klassen des Reflection-Paketes [`java.lang.reflect`](#).

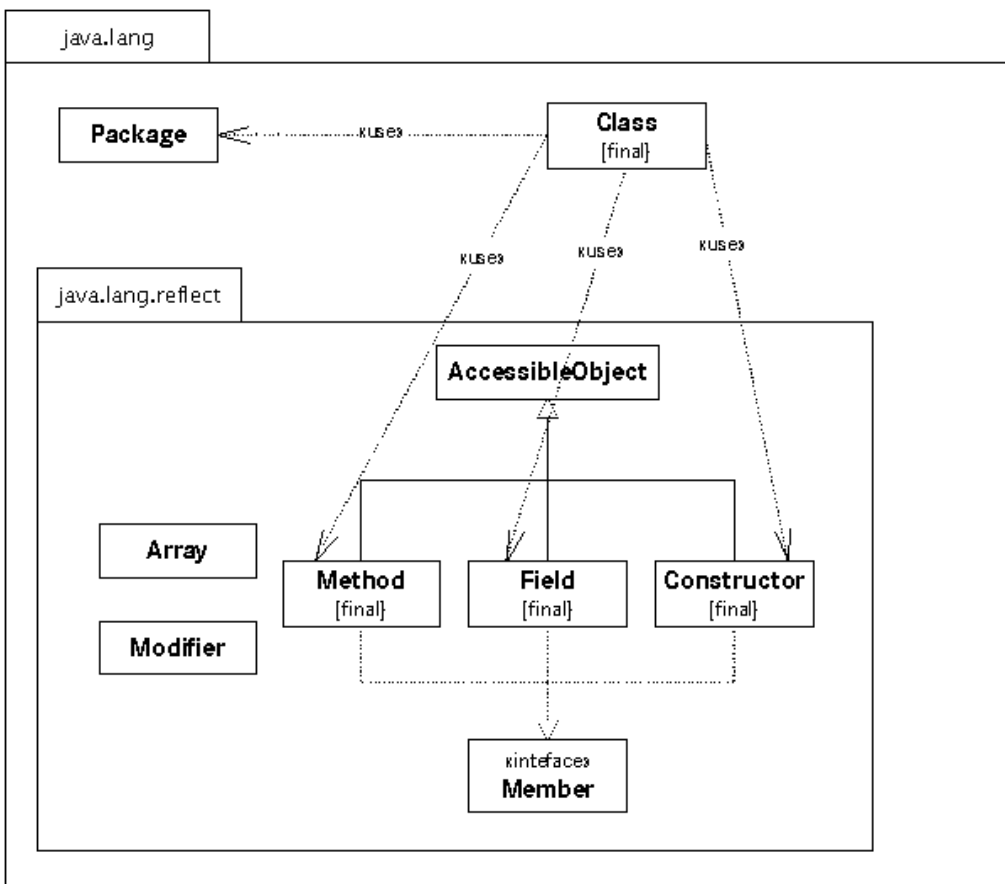


Abbildung 59: Ausschnitt aus dem Reflection-Paket und Abhängigkeiten zu anderen Klassen (click on image to enlarge!)

Die Reflection-API ist als Subpaket von `java.lang` organisiert. Ihre Klassen dienen lediglich zur Inspektion existierender Klassen einer Java-Laufzeit-Umgebung. Daher sind die Konstruktoren der von `AccessibleObject` abgeleiteten Klassen, sowie der der Klasse `Array`, als `private` deklariert; Ausprägungen dieser Klassen können ausschließlich durch die virtuelle Maschine erzeugt werden.

Im Folgenden werden zunächst die Methoden zur Ermittlung verschiedenster Informationen am [konkreten Codebeispiel](#) vorgestellt.

Die verschiedenen Methoden zur Einflußnahme auf bereits erzeugte Metainformationsobjekte werden [im Anschluß](#) diskutiert.

Ermittlung Klassen- und Schnittstellen-bezogener Information

Klassen und Schnittstellen werden durch die Reflection-API gleichbehandelt. Die Unterscheidung kann durch Methode `isInterface` auf jedem `Class`-Objekt getroffen werden.

Ferner stehen zur Verfügung:

- **Modifier** -- [getModifiers](#)
Der Aufruf liefert die verschiedenen Details als `int`-Wert, der die gesetzten Modifier Bit-codiert enthält. Zur Entschlüsselung der einzelnen Modifier sollten die Methoden der Reflection-Klasse `Modifier` eingesetzt werden, um auch hinsichtlich zukünftiger Java-Spracherweiterungen stabilen Code zu produzieren.
[Aufruf im Beispiel](#)
Die [Umsetzung im Beispiel](#) nutzt die Implementierung der `toString()`-Methode der Klasse `Modifier`, welche die in der [Java Language Specification](#) definierte kanonische Repräsentation zurückliefert.
- **Komponententyp-Ermittlung** -- [getComponentType\(\)](#)
Ermöglicht die Ermittlung des zugrundeliegenden Typen falls es sich bei der Klasse um einen Array handelt. Bei geschachtelten Arrays ist diese Methode u.U. mehrfach anzuwenden.
[Verwendung im Beispiel](#)
- **Intern deklarierte Klassen und Schnittstellen** -- [getDeclaredClasses\(\)](#)
Liefert die deklarierten (nicht anonymen) [inneren Klassen](#) und inneren Schnittstellen als `Class`-Array zurück.
[Verwendung im Beispiel](#)
- **Name** -- [getName\(\)](#)
Liefert den vollqualifizierten Namen der Klasse oder Schnittstelle.

Verwendung im Beispiel

- Paketzugehörigkeit -- [getPackage](#)

Liefert das definierende Paket der Klasse als Ausprägung der Klasse [Package](#). Bei Klassen im Vorgabepaket, d.h. ohne explizite package-Definition, wird null zurückgegeben.

Verwendung im Beispiel**Ermittlung Attribut-bezogener Informationen**

Der Zugriff auf Attribute und Attribut-spezifische Informationen -- in der Java-Terminologie Felder genannt -- (wie Namen, Sichtbarkeitsbereich und Typ) wird durch die Methoden [getField\(String\)](#), für ein namentlich bekanntes Feld, und [getFields\(\)](#) für *alle* Attribute einer Klasse, realisiert. In beiden Fällen wird genau ein, bzw. ein Array von Objekten der Reflection-Klasse [Field](#) zurückgeliefert.

Beide Methoden berücksichtigen dabei *alle* Attribute/Felder einer Klasse, einschließlich der von der Superklasse und deren Superklassen ererbten. Zur Ermittlung der direkt auf dem betrachteten Class-Objekt definierten Attribute können Methoden [getDeclaredField\(String\)](#) bzw. [getDeclaredFields\(\)](#) eingesetzt werden.

Verwendung im Beispiel

[Field](#)-Objekte bieten Zugriff auf die verschiedenen deklarativen Charakteristika eines Attributs:

- Modifier ([siehe Kapitel 2](#)) -- [getModifiers\(\)](#).
Der Aufruf liefert die verschiedenen Details als int-Wert, der die gesetzten Modifier Bit-codiert enthält. Zur Entschlüsselung der einzelnen Modifier sollten die Methoden der Reflection-Klasse [Modifier](#) eingesetzt werden, um auch hinsichtlich zukünftiger Java-Spracherweiterungen stabilen Code zu produzieren.
Die [Verwendung im Beispiel](#) nutzt die Implementierung der [toString\(\)](#)-Methode der Klasse [Modifier](#), welche die in der [Java Language Specification](#) definierte kanonische Repräsentation zurückliefert.
- Typ (einer der [Primitivtypen](#), ein Array, eine beliebige zugreifbare Klasse oder eine Schnittstelle) -- [getType\(\)](#).
Die Methode liefert ein Objekt vom Typ `Class` zurück, auch wenn die Primitivtypen nicht als (first class) Objekte realisiert sind.
Zur Ermittlung des tatsächlichen Typs sind weitere Operationen auf dem Class-Objekt notwendig. [isArray\(\)](#) klärt ob es sich bei dem Attributtypen um einen Array-Typ handelt. Da Java mehrdimensionale Arrays als geschachtelte eindimensionale Pendants realisiert muß in einem solchen Falle per [getComponentType\(\)](#) weiternavigiert werden. Diese Methode liefert wiederum ein Class-Objekt zurück.
Der vollqualifizierte (d.h. im Falle der Existenz um den durch das Paket gebildeten Namensraum angereicherte) Name eines Typen wird durch [getName\(\)](#) zurückgeliefert.
Die Methode [printAttribute](#) des Beispiels zeigt die Ermittlung des Typs durch die vorgestellten Methoden im Überblick.
- Name -- [getName\(\)](#).
Liefert den Attributnamen als String.

Ermittlung Operationen-bezogener Information

Anmerkung: Zwar spricht die Java-Reflection-API hier konsequent von *Methoden*, bietet jedoch keine Zugriffsmöglichkeiten auf den Implementierungskörper einer Operation. Ermittelt werden daher nur die Signatur, die Sichtbarkeitsmodifier sowie der Rückgabetyt und, falls existent, im Ausführungsverlauf ausgelöste Ausnahmeereignisse; mithin: die *Operation*.

Die Reflection-API trennt Konstruktoren und sonstige Operationen voneinander. Daher existieren mit [getConstructor\(Class\[\]\)](#) und [getConstructors\(\)](#) bzw. [getMethod\(String, Class\[\]\)](#) und [getMethods\(\)](#) jeweils eigene Methoden zu ihrer Ermittlung.

Zurückgeliefert wird jedoch in allen Fällen genau ein, bzw. ein Array von, [Method](#)-Objekt(en).

- Modifier ([siehe Kapitel 2](#)) -- [getModifiers\(\)](#)
Der Aufruf liefert die verschiedenen Details als int-Wert, der die gesetzten Modifier Bit-codiert enthält. Zur Entschlüsselung der einzelnen Modifier sollten die Methoden der Reflection-Klasse [Modifier](#) eingesetzt werden, um auch hinsichtlich zukünftiger Java-Spracherweiterungen stabilen Code zu produzieren.
Die [Umsetzung im Beispiel](#) nutzt die Implementierung der [toString\(\)](#)-Methode der Klasse [Modifier](#), welche die in der [Java Language Specification](#) definierte kanonische Repräsentation zurückliefert.
- Übergabeparameter -- [getParameterTypes\(\)](#)
Liefert einen Array von Class-Objekten, die die formalen Parameter der Operation repräsentieren.
[Verwendung im Beispiel](#)

- Rückgabetyp (einer der [Primitivtypen](#), ein Array, eine beliebige zugreifbare Klasse oder eine Schnittstelle) -- [getReturnType\(\)](#).

[Verwendung im Beispiel](#)

- Auslösbare Ausnahmeereignisse -- [getExceptionTypes\(\)](#)

Liefert einen Array von Class-Objekten der deklarierten Exceptions der Operation.

[Verwendung im Beispiel](#)

```
(1)import java.lang.reflect.Constructor;
(2)import java.lang.reflect.Method;
(3)import java.lang.reflect.Modifier;
(4)import java.lang.reflect.Member;
(5)import java.lang.reflect.Field;
(6)import java.lang.ClassNotFoundException;
(7)
(8)public class ShowClass {
(9)    public static void main(String[] args) {
(10)        try {
(11)            printClass( Class.forName(args[0]) );
(12)        } catch (ClassNotFoundException cnfe) {
(13)            System.out.println("cannot find class "+args[0]);
(14)        } //catch
(15)    } //main()
(16)// -----
(17)    public static void printClass(Class c) {
(18)        Package pck = c.getPackage();
(19)        if ( pck != null )
(20)            System.out.println("package "+pck.getName()+";" );
(21)
(22)        if (c.isInterface()) {
(23)            System.out.print(Modifier.toString(c.getModifiers()) + " "+ c.getName
(24)());
(25)        } else
(26)            System.out.print(Modifier.toString(c.getModifiers()) + " class " +
(27)                c.getName() +
(28)                " extends " + c.getSuperclass().getName());
(29)
(30)        Class[] interfaces = c.getInterfaces();
(31)        if ((interfaces != null) && (interfaces.length > 0)) {
(32)            if (c.isInterface())
(33)                System.out.println(" extends ");
(34)            else
(35)                System.out.print(" implements ");
(36)            for(int i = 0; i < interfaces.length; i++) {
(37)                if (i > 0)
(38)                    System.out.print(", ");
(39)                System.out.print(interfaces[i].getName());
(40)            } //for
(41)            System.out.println("\n{");
(42)
(43)            Constructor[] constructors = c.getDeclaredConstructors();
(44)            if ( constructors.length > 0 ) {
(45)                System.out.println("    //Constructors");
(46)                for(int i = 0; i < constructors.length; i++)
(47)                    printOperation(constructors[i]);
(48)            } //if
(49)            constructors = null;
(50)
(51)            Class[] classes = c.getDeclaredClasses();
(52)            if (classes.length > 0) {
(53)                System.out.println("    //Inner Classes");
(54)                for (int i=0; i < classes.length; i++) {
(55)                    if (classes[i].isInterface() == false ) {
(56)                        printClass(classes[i] );
(57)                        System.out.print("\n");
(58)                    } //if
(59)                } //for
(60)                System.out.println("    //Inner Interfaces");
(61)                for (int i=0; i < classes.length; i++) {
(62)                    if (classes[i].isInterface() == true ) {
(63)                        printClass(classes[i] );
(64)                        System.out.print("\n");
```

```

(65)                } //if
(66)                } //for
(67)            } //if
(68)            classes = null;
(69)
(70)            Field[] fields = c.getDeclaredFields();
(71)            if ( fields.length > 0 ) {
(72)                System.out.println("    //Attributes");
(73)                for(int i = 0; i < fields.length; i++)
(74)                    printAttribute(fields[i]);
(75)            } //if
(76)            fields = null;
(77)
(78)
(79)            Method[] operations = c.getDeclaredMethods();
(80)            if ( operations.length > 0 ) {
(81)                System.out.println("    //Operations");
(82)                for(int i = 0; i < operations.length; i++)
(83)                    printOperation(operations[i]);
(84)            } //if
(85)            operations = null;
(86)
(87)            System.out.print("} //end ");
(88)            if ( c.isInterface() )
(89)                System.out.print("interface ");
(90)            else
(91)                System.out.print("class ");
(92)            System.out.print( c.getName() );
(93)        } //printClass(Class)
(94)// -----
(95)        public static String typename(Class t) {
(96)            String brackets = "";
(97)            while(t.isArray()) {
(98)                brackets += "[";
(99)                t = t.getComponentType();
(100)            } //while
(101)            return t.getName() + brackets;
(102)        } //typename(Class)
(103)// -----
(104)        public static String modifiers(int m) {
(105)            if (m == 0)
(106)                return "";
(107)            else
(108)                return Modifier.toString(m) + " ";
(109)        } //modifiers(int)
(110)// -----
(111)        public static void printAttribute(Field f) {
(112)            System.out.print(" " + modifiers(f.getModifiers()) + typename(f.
(113)                getType()) + " " + f.getName() + ";    //" + f.getType().getName() + " is ");
(114)
(115)            if ( (f.getType()).isInterface() )
(116)                System.out.println("an interface");
(117)            else {
(118)                if ( f.getType().isPrimitive() )
(119)                    System.out.println("a primitive type");
(120)                else {
(121)                    if ( f.getType().isArray() )
(122)                        System.out.println("an array");
(123)                    else
(124)                        System.out.println("a class");
(125)                } //else
(126)            } //else
(127)        } //printAttribute(Field)
(128)        public static void printOperation(Member member) {
(129)            Class returnType=null, parameters[], exceptions[];
(130)
(131)            if (member instanceof Method) {
(132)                Method m = (Method) member;
(133)                returnType = m.getReturnType();
(134)                parameters = m.getParameterTypes();
(135)                exceptions = m.getExceptionTypes();

```

```

(136)    } else {
(137)        Constructor c = (Constructor) member;
(138)        parameters = c.getParameterTypes();
(139)        exceptions = c.getExceptionTypes();
(140)    } //else
(141)
(142)    System.out.print(" " + modifiers(member.getModifiers()) + ((returntype!
= null)? typename(returntype)+" " : "") + member.getName() + "(");
(143)    for(int i = 0; i < parameters.length; i++) {
(144)        if (i > 0)
(145)            System.out.print(", ");
(146)
(147)        System.out.print(typename(parameters[i]));
(148)    } //for
(149)
(150)    System.out.print(")");
(151)
(152)    if (exceptions.length > 0)
(153)        System.out.print(" throws ");
(154)
(155)    for(int i = 0; i < exceptions.length; i++) {
(156)        if (i > 0)
(157)            System.out.print(", ");
(158)
(159)        System.out.print(typename(exceptions[i]));
(160)    } //for
(161)
(162)    System.out.println(";");
(163) } //printOperation(Member)
(164)} //class ShowClass
(165)
(166)// -----
(167)interface TestInterface {
(168)} //interface TestInterface
(169)
(170)class TestClass {
(171)    public        TestInterface i1;
(172)    private        TestClass c1;
(173)    protected    int p1;
(174)    int[]          ta1;
(175)    int[][]         tam1;
(176)
(177)    class Inner {
(178)        int i;
(179)        class InnerInner {
(180)            int j;
(181)        } //class InnerInner
(182)    } //class Inner
(183)    interface InnerInterface {
(184)    } //interface InnerInterface
(185)
(186)    private TestClass testc() {
(187)        return new TestClass() {
(188)            public void hello() {
(189)                System.out.println("overridden test!");
(190)            } //hello()
(191)        }; //class TestClass
(192)    } //test()
(193)} //class TestClass

```

Beispiel 85: Erfragen verschiedener Klassen-spezifischer Informationen durch die Reflection-API [ShowClass.java](#)

BildschirmAusgabe:

```

class TestClass extends java.lang.Object
{
    //Constructors
    TestClass();
    //Inner Classes

```

```

class TestClass$Inner extends java.lang.Object
{
    //Constructors
    TestClass$Inner(TestClass);
    //Inner Classes
    class TestClass$Inner$InnerInner extends java.lang.Object
    {
        //Constructors
        TestClass$Inner$InnerInner(TestClass$Inner);
        //Attributes
        int j;        //int is a primitive type
        final TestClass$Inner this$1;    //TestClass$Inner is a class
    } //end class TestClass$Inner$InnerInner
        //Inner Interfaces
        //Attributes
        int i;        //int is a primitive type
        final TestClass this$0;    //TestClass is a class
    } //end class TestClass$Inner
        //Inner Interfaces
    abstract static interface TestClass$InnerInterface
    {
    } //end interface TestClass$InnerInterface
        //Attributes
        public TestInterface i1;    //TestInterface is an interface
        private TestClass c1;    //TestClass is a class
        protected int p1;    //int is a primitive type
        int[] ta1;    //[I is an array
        int[][] tam1;    //[I is an array
        //Operations
        private TestClass testc();
    } //end class TestClass

```

Das Ausgabe enthält, erwartungsgemäß, die implizite Superklasse `java.lang.Object`, welche durch den Übersetzer automatisch gesetzt wird, als explizite Angabe nach dem `extends`-Schlüsselwort.

Die inneren Klassen erscheinen in ihrer JVM-internen Namensgebung mit dem trennenden Dollarsymbol `$`. Auffallend ist die explizite Wiedergabe der `this`-Variable und des automatisch generierten Konstruktors mit dem Typ der äußeren Klasse als Übergabeparameter. Für Array-Typen erfolgt die Benennung ebenfalls gemäß der JVM-internen Konvention.

Erzeugung neuer Objekte und Modifikationen bestehender mit der Reflection API

Neben den bisher vorgestellten Mechanismen zur Introspektion bestehender Strukturen kann die Reflection API auch zur Erzeugung neuer Objekte und Operation auf diesen eingesetzt werden. Hierfür stehen neben Methoden zum lesenden und schreibenden Zugriff auf Attribute auch Möglichkeiten zur Ausführung von Methoden auf den Objekten zur Verfügung.

Erzeugung von Objekten

Die Erzeugung neuer Objekte aus bestehenden Class-Ausprägungen war [bereits mit Methoden der Class-Klasse möglich](#), sofern zur Objekterzeugung der parameterlose Standardkonstruktor verwendet wird.

Zur Instanziierung neuer Objekte unter Nutzung eines bestehenden parametrisierten Konstruktors ist die Verwendung der Reflection-Klasse [Constructor](#) unabdingbar.

Der Aufruf eines parametrisierten Konstruktors vollzieht sich in drei Schritten:

1. Ermittlung des Class-Objekts für die neu zu erzeugende Instanz
2. Erzeugung eines neuen Constructor-Objekts mit der [getConstructor\(Class\[\]\)](#)-Methode der Klasse `Class`
3. Objekterzeugung durch die [newInstance\(Object\[\]\)](#)-Methode der Constructor-Klasse

```

(1)import java.lang.reflect.Constructor;
(2)import java.lang.reflect.InvocationTargetException;
(3)import java.awt.Rectangle;
(4)
(5)class SampleInstance {
(6)    public static void main(String[] args) {
(7)        Rectangle rectangle;
(8)        Class rectangleDefinition;
(9)        Class[] intArgsClass = new Class[] {int.class, int.class};
(10)        Integer height = new Integer(12);
(11)        Integer width = new Integer(34);
(12)        Object[] intArgs = new Object[] {height, width};
(13)        Constructor intArgsConstructor;
(14)
(15)        try {
(16)            rectangleDefinition = Class.forName("java.awt.Rectangle");
(17)            intArgsConstructor = rectangleDefinition.getConstructor
(intArgsClass);
(18)
(19)            System.out.println ("Constructor: " + intArgsConstructor.toString());
(20)            Object object = null;
(21)
(22)            object = intArgsConstructor.newInstance(intArgs);
(23)            System.out.println ("Object: " + object.toString());
(24)            rectangle = (Rectangle) object;
(25)        } catch (ClassNotFoundException e) {
(26)            System.out.println("A ClassNotFoundException occurred!\n"+e.toString()
+"\n"+e.getMessage() );
(27)        } catch (NoSuchMethodException e) {
(28)            System.out.println("A NoSuchMethodException occurred!\n"+e.toString()
+"\n"+e.getMessage() );
(29)        } catch (InstantiationException e) {
(30)            System.out.println("An InstantiationException occurred!\n"+e.toString()
+"\n"+e.getMessage() );
(31)        } catch (IllegalAccessException e) {
(32)            System.out.println("An IllegalAccessException occurred!\n"+e.toString()
+"\n"+e.getMessage() );
(33)        } catch (IllegalArgumentException e) {
(34)            System.out.println("An IllegalArgumentException occurred!\n"+e.
toString()+"\n"+e.getMessage() );
(35)        } catch (InvocationTargetException e) {
(36)            System.out.println("An InvocationTargetException occurred!\n"+e.
toString()+"\n"+e.getMessage() );
(37)        } //catch
(38)    } //main()
(39)} //class SampleInstance

```

Beispiel 86: Erzeugung eines AWT-Rechtecks über einen parametrisierten Konstruktor mit der Reflection-API [SampleInstance.java](#)

Bildschirmausgabe:

```

Constructor: public java.awt.Rectangle(int,int)
Object: java.awt.Rectangle[x=0,y=0,width=12,height=34]

```

Das Beispiel spiegelt die drei Erstellungsschritte wieder:

Zunächst wird über den Namen (Aufruf: `forName(...)`) das zur Klasse gehörige Class-Objekt erfragt.

Die Methode `getConstructor(...)` liefert das Konstruktorenobjekt welches die als Parameter übergebene Signatur aufweist.

Instanziiert wird das neue Objekt durch die Methode `newInstance`, ausgeführt auf dem Constructor-Objekt mit den definierten Übergabeparametern.

Attributzugriff

... geschieht über Objekte der Klasse `Field`.

Für alle Primitivtypen existieren lesende Methoden, die einheitlich mit `getT(Object)` signiert sind.

`T` ist einer der Primitivtypen ist. Der Rückgabebetyp ist jeweils `T`.

Werte von objektartigen Attributen können mittels `get(Object)` erfragt werden; zurückgegeben wird hier eine Ausprägung von `Object`.

Symmetrisch sind die schreiben Zugriffe als `setT(Object, T)` realisiert. Mit `set(Object, Object)` steht eine Methode zur Ablage objektartiger Attribute zur Verfügung.

Das nachfolgende Beispiel zeigt die Verwendung der Lese- und Schreibmethoden.

```
(1)import java.lang.reflect.Field;
(2)
(3)public class AttributeAccess {
(4)    public static void main(String[] args) {
(5)        Test o1 = new Test();
(6)        o1.setI(42);
(7)
(8)        System.out.println( "i="+o1.getI() );
(9)
(10)       try {
(11)           Field iAttribute = (o1.getClass()).getField("i");
(12)           System.out.println( "i="+iAttribute.getInt(o1) );
(13)
(14)           iAttribute.setInt(o1, 50 );
(15)           System.out.println( "i="+iAttribute.getInt(o1) );
(16)       } catch (NoSuchFieldException e) {
(17)           System.out.println("A NoSuchFieldException occurred!\n"+e.
toString()+"\n"+e.getMessage() );
(18)           e.printStackTrace();
(19)       } catch (IllegalAccessException e) {
(20)           System.out.println("A IllegalAccessException occurred!\n"+e.
toString()+"\n"+e.getMessage() );
(21)           e.printStackTrace();
(22)       } //catch
(23)    } //main()
(24)} //class AttributeAccess
(25)
(26)class Test {
(27)    public int i;
(28)
(29)    public void setI(int i) {
(30)        this.i = i;
(31)    } //setI()
(32)
(33)    public int getI() {
(34)        return i;
(35)    } //getI()
(36)} //class Test
```

Beispiel 87: Attributzugriff mit Methoden der Reflection-Klasse `Field` [AttributeAccess.java](#)

Bildschirmausgabe:

```
i=42
i=42
i=50
```

Ausführen von Methoden

Die Klasse `Method` weist neben den bisher vorgestellten Funktionalitäten zur Inspektion auch mit der Methode `invoke(Object, Object[])` einen generischen Mechanismus zur Ausführung beliebiger Methoden auf.

Das folgende Beispiel zeigt den Aufruf der überladenen Methode `hello` über die `invoke`-Methode der Reflection-API.


```

(1)import java.lang.reflect.Method;
(2)import java.lang.reflect.InvocationTargetException;
(3)
(4)public class MethodExecution {
(5)    public static void main(String[] args) {
(6)        test o1 = new test();
(7)
(8)        try {
(9)            Method helloMethod = (o1.getClass()).getMethod("hello",
null);
(10)            helloMethod.invoke(o1, null);
(11)
(12)            Class[] formalParameters = { String.class };
(13)            helloMethod = (o1.getClass()).getMethod("hello",
formalParameters );
(14)            Object[] argumentList = { "stranger" };
(15)            helloMethod.invoke(o1, argumentList);
(16)        } catch (NoSuchMethodException e) {
(17)            System.out.println("A NoSuchMethodException occurred!\n"+e.
toString()+"\n"+e.getMessage() );
(18)            e.printStackTrace();
(19)        } catch (IllegalAccessException e) {
(20)            System.out.println("A NoSuchMethodException occurred!\n"+e.
toString()+"\n"+e.getMessage() );
(21)            e.printStackTrace();
(22)        } catch (InvocationTargetException e) {
(23)            System.out.println("An InvocationTargetException occurred!
\n"+e.toString()+"\n"+e.getMessage() );
(24)            e.printStackTrace();
(25)        } //catch
(26)    } //main()
(27)} //class MethodExecution
(28)
(29)class Test {
(30)    public void hello() {
(31)        System.out.println("simple hello!");
(32)    } //hello()
(33)
(34)    public void hello(String name) {
(35)        System.out.println("hello "+name);
(36)    } //hello()
(37)} //class Test

```

Beispiel 88: Ausführung zweiter Methoden durch die Reflection-API [MethodExecution.java](#)

BildschirmAusgabe:

```

simple hello!
hello stranger

```

Weitere Anwendungsbeispiele

Unter Einsatz der Reflection-API lassen sich generische Programme entwickeln, die auf beliebigen Typen operieren. Mit diesem Mechanismus läßt sich ein Verhalten vergleichbar der parametrischen Polymorphie der C++-Templates nachbilden.

```

(1)import java.lang.reflect.Array;
(2)
(3)public class GrowableArrayTest {
(4)    public static void main(String[] args) {
(5)        int[] ia = {1};
(6)        System.out.println("ia consists of "+ia.length+" elements");
(7)        ia = (int[]) arrayGrow(ia);
(8)        System.out.println("ia consists of "+ia.length+" elements");
(9)
(10)        int[] ib = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
(11)        System.out.println("ib consists of "+ib.length+" elements");

```

```

(12)         ib = (int[]) arrayGrow(ib);
(13)         System.out.println("ib consists of "+ib.length+" elements");
(14)
(15)         Person[] pa = new Person[3];
(16)         pa[0] = new Person("hans");
(17)         pa[1] = new Person("franz");
(18)         pa[2] = new Person("fritz");
(19)         System.out.println("pa consists of "+pa.length+" elements");
(20)         pa = (Person[]) arrayGrow(pa);
(21)         System.out.println("pa consists of "+pa.length+" elements");
(22)     } //main()
(23)
(24)     static Object arrayGrow(Object oldArray) {
(25)         Class arrayClass = oldArray.getClass();
(26)         if ( !arrayClass.isArray() )
(27)             return null;
(28)
(29)         Class componentType = arrayClass.getComponentType();
(30)
(31)         int oldLength = Array.getLength(oldArray);
(32)         int newLength = ( ((oldLength * 1.1) > oldLength + 1) ? (int)
(33)         (oldLength * 1.1) : (oldLength + 1) );
(34)         Object newArray = Array.newInstance(componentType, newLength );
(35)
(36)         System.arraycopy(oldArray, 0, newArray, 0, oldLength);
(37)         return newArray;
(38)     } //arrayGrow()
(39)} //class growableArrayTest
(40)
(41)class Person {
(42)    protected String name;
(43)
(44)    public Person (String name) {
(45)        this.name = name;
(46)    } //constructor
(47)} //class Person

```

Beispiel 89: Ein wachsender Array [GrowableArrayTest.java](#)

BildschirmAusgabe:

```

ia consists of 1 elements
ia consists of 2 elements
ib consists of 20 elements
ib consists of 22 elements
pa consists of 3 elements
pa consists of 4 elements

```

Das Beispiel implementiert dynamische, durch expliziten Methodenaufruf, wachsende Arrays. Nach Ermittlung des Class-Objekts (durch `getClass()`) und des Komponententyps des Arrays (`getComponentType()`) wird ein zweiter -- der neue -- Array durch die `newInstance`-Methode der Klasse `Array` erzeugt. Als Übergabeparameter wird der Komponententyp und die Länge des neuen Arrays übergeben. Die Länge des neuen Arrays entspricht der des Alten, um zehn Prozent, jedoch mindestens ein Element, erhöht.

Abschließend wird der Inhalt des alten in den neu erzeugten Array kopiert. Hierzu wird die sehr performant ausgeführte native Methode `System.arraycopy` eingesetzt.

Durch die Konstruktion des Method-Objekts und der `invoke`-Methode läßt sich auch das aus C/C++ bekannte Verhalten der **Funktionszeiger** nachbilden.

Das Beispiel zeigt zweimaligen Aufruf derselben Methoden `printTable`. Im Körper der Methode wird jeweils die als Parameter übergebene Methode ausgeführt.

Im ersten Aufruf wird das Quadrat der Zahlen von Eins bis Zehn gebildet, im Zweiten hingegen die Wurzel gezogen.

```

(1)import java.lang.reflect.Method;
(2)import java.lang.reflect.InvocationTargetException;
(3)
(4)public class MethodPointerTest {
(5)    public static void main(String[] args) {
(6)        try {
(7)            printTable(1,10, MethodPointerTest.class.getMethod("square",
new Class[] {double.class} ));
(8)            printTable(1,10, java.lang.Math.class.getMethod("sqrt", new
Class[] { double.class} ));
(9)        } catch (NoSuchMethodException e) {
(10)            System.out.println("A NoSuchMethodException occurred!\n"+e.
toString()+"\n"+e.getMessage() );
(11)        } //catch
(12)    } //end main()
(13)
(14)    public static double square (double x) {
(15)        return (x*x);
(16)    } //square()
(17)
(18)    public static void printTable(double from, double to, Method f) {
(19)        System.out.println(f);
(20)        for (double x=from; x <= to; x++) {
(21)            System.out.print( x );
(22)            try {
(23)                Object[] args = { new Double(x) };
(24)                Double d = (Double) f.invoke(null,args);
(25)                double y = d.doubleValue();
(26)                System.out.println(" | "+y);
(27)            } catch (InvocationTargetException e) {
(28)                System.out.println("An InvocationTargetException
occurred!\n"+e.toString()+"\n"+e.getMessage() );
(29)            } catch (IllegalAccessException e) {
(30)                System.out.println("An IllegalAccessException
occurred!\n"+e.toString()+"\n"+e.getMessage() );
(31)            } //catch
(32)        } //for
(33)    } //printTable()
(34)} //class MethodPointerTest

```

Beispiel 90: Verweise auf Methoden in Java [MethodPointerTest.java](#)

BildschirmAusgabe:

```

public static double MethodPointerTest.square(double)
1.0 | 1.0
2.0 | 4.0
3.0 | 9.0
4.0 | 16.0
5.0 | 25.0
6.0 | 36.0
7.0 | 49.0
8.0 | 64.0
9.0 | 81.0
10.0 | 100.0
public static strictfp double java.lang.Math.sqrt(double)
1.0 | 1.0
2.0 | 1.4142135623730951
3.0 | 1.7320508075688772
4.0 | 2.0
5.0 | 2.23606797749979
6.0 | 2.449489742783178
7.0 | 2.6457513110645907
8.0 | 2.8284271247461903
9.0 | 3.0
10.0 | 3.1622776601683795

```

▲ 3.2.7 Abstract Windowing Toolkit (AWT)

Das AWT stellt den wohl interessantesten Teil eines (interaktiven) Java-Programmes dar -- die Benutzerinteraktion durch eine graphische Oberfläche.

Ziel der AWT-Entwicklung war es, die verschiedenen graphischen Primitiven wie Buttons, Menüs etc. *plattformunabhängig* anzubieten.

Zur Realisierung des *Look and Feel* bieten sich vier verschiedene Ansätze an:

- Umsetzung ausschließlich der gemeinsamen graphischen Komponenten aller Zielplattformen. Hierdurch entsteht automatisch ein GUI des kleinsten gemeinsamen Nenners, welches -- insbesondere wenn man kleinste Endgeräte wie PDAs und Mobiltelefone betrachtet -- praktisch nur schwerlich einsetzbar sein dürfte.
- Umsetzung absolut aller Komponenten der verschiedenen Zielplattformen auf jeder Zielplattform. Hierdurch entsteht automatisch ein GUI das die Vereinigungsmenge aller betrachteten Plattformen abbildet. Differenzieren die betrachteten Plattformen stark in ihrer Leistungsfähigkeit, so zieht dies mitunter sehr hohen Aufwand in der späteren Umsetzung (Emulation) der einzelnen Primitive auf der realen Plattform nach sich.
- Kapselung der plattformspezifischen API durch die neue Zugriffsschicht. Hierdurch steht dem Anwender auf der realen Plattform die dort übliche graphische Schnittstelle in Aussehen und Verhalten zur Verfügung. Alternativ könnte auch eine vollständig neue GUI definiert werden, die dann auf allen realen Plattformen unterstützt wird. Diesen Ansatz verfolgt die Java 2 Plattform mit *SWING*.
- Bündelung und Umsetzung einer Menge *sinnvoller*, gängiger Konzepte. Hierdurch steht auf der realen Zielplattform immer dieselbe GUI zur Verfügung, die je nach tatsächlicher Ausgestaltung der Plattform mehr oder minder aufwendig zu implementieren ist.

Das AWT bietet neben den graphischen Primitivoperationen zum Zeichnen von einfachen geometrischen Objekten wie Linien, Rechtecken, Kreisen; Fülloperationen und Methoden zur Textausgabe einen Mechanismus zur ereignisbasierten Ablaufsteuerung an, der es erlaubt auf externe Ereignisse wie Mauseingaben zu reagieren. Ferner bietet sie die weitgehend bekannten GUI-Grundelemente wie Fenster, Dialogboxen, Menüs, ... an. Zur Entwicklung komplexerer Anwendungen steht Funktionalität zur Graphikbearbeitung und Audiowiedergabe zur Verfügung.

... und SWING?

Mit der Java Version 1.1 wurde das AWT vollständig überarbeitet, insbesondere die oftmals als Achillesferse praktisch verwendbarer Applikationen empfundene Ereignisbehandlung (engl. *event handling*). Seither verwirklicht auch Java das bereits von anderen Oberflächensystemen wie NextStep oder Windows NT bekannte *Delegation Based Event Handling*. Der Terminus beschreibt die Möglichkeit der AWT externe Ereignisse (Mausclicks, Fenster-Schließen, ...) an beliebige Objekte zur Behandlung weiterzureichen.

Ebenfalls mit der JDK-Version 1.1 wurde eine zweite Bibliothek zur Oberflächenprogrammierung vorgestellt: *SWING*. Sie geht deutlich über den in der AWT realisierten Funktionsumfang hinaus, und wird daher heute überwiegend zur Implementierung professioneller Anwendungen herangezogen.

Vorwärtsreferenz: Sie wird in Kapitel [3.2.8 Swing](#) beschrieben.

Struktur des AWT

Das gesamte AWT ist im Standard-API-Paket [java.awt](#) zusammengefaßt. Es stellt die grundlegenden graphischen Primitive zur Verfügung:

- Auswahlfelder [List](#)
enthält auswählbare Elemente.
- Beschriftungstexte [Label](#)
Einzeilige Beschriftung.
- Bildlaufleisten (scroll bars)
Verschieberegler zum Scrollen.
- [Choice](#)
Drop-down-Liste oder Optionsmenü.
- [FileDialog](#)
Dialogbox zur Auswahl von Dateien.
- leere Komponente [Canvas](#)
Kann zur Anzeige eigener Zeichnungen verwendet werden.
- Schaltflächen [Button](#)
Graphische Schaltfläche, die vom Benutzer angeklickt werden kann.
- Textbereiche (text area)
Mehrzeiliges Textfeld für Benutzereingaben und zur Textanzeige.

- Textfelder (text field)
Einzeliges Textfeld für Benutzereingaben.
- Umschaltknopf [Checkbox](#)
kann selektiert oder deselektiert werden.

Als Menü-Komponenten stehen zur Verfügung:

- [Menu](#)
Pull-down oder frei platzierbares Menü (*Abreißmenü*).
- [CheckboxMenuItem](#)
Umschaltknopf in einem Menü.
- [MenuBar](#)
... kann Pull-down-Menüs aufnehmen.
- [MenuItem](#)
Menü-Schaltfläche.
- [PopupMenu](#)
... öffnet sich automatisch bei Mauskontakt.

Die dargestellten Komponenten dienen zunächst zum Aufbau der Benutzeroberfläche (buttons, Listen, etc.) oder zur Strukturierung der Fenster (scroll bars, menus, etc.). Die dritte Kategorie bilden die Zeichenbereiche.

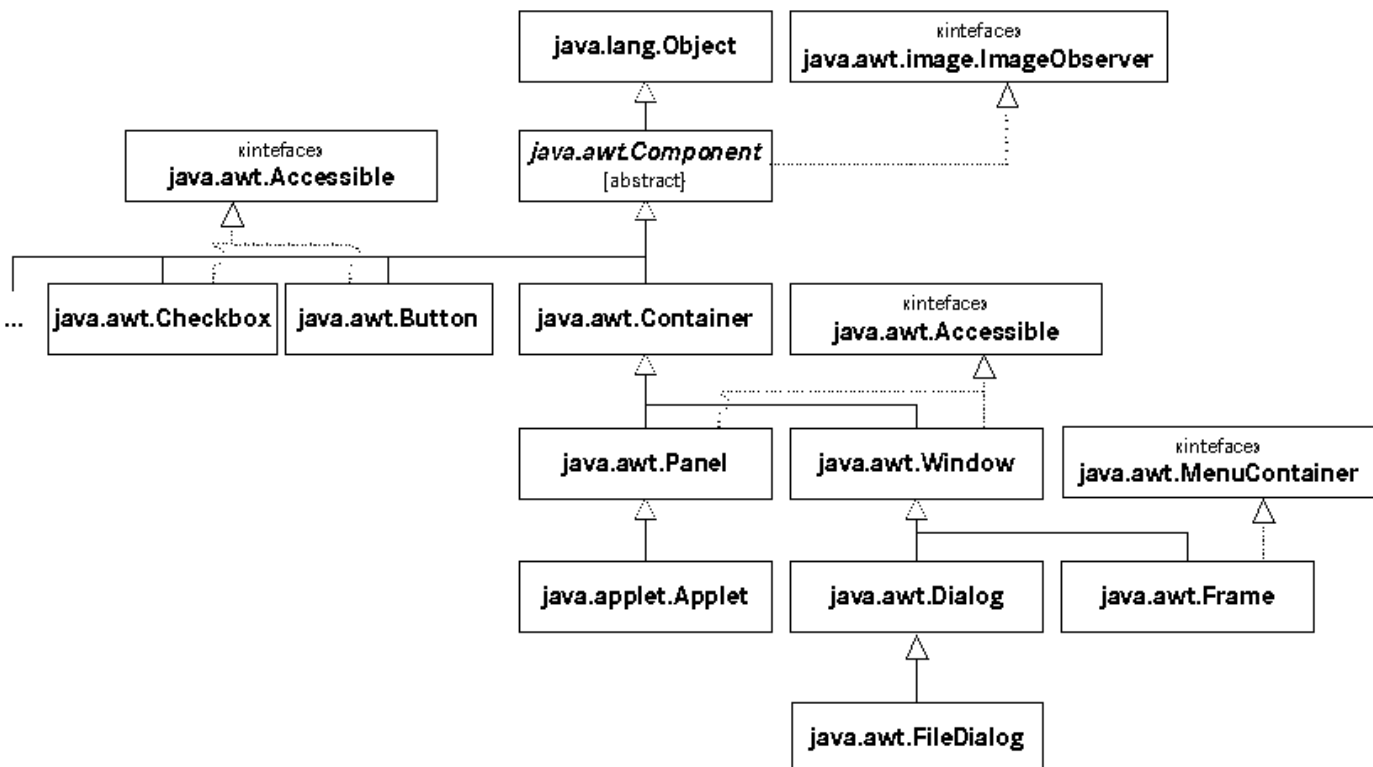


Abbildung 60: Ausschnitt aus der AWT-Klassenhierarchie (click on image to enlarge!)

Ausgangspunkt jeder AWT-basierten Applikation ist die -- bereits aus der Applet-Hierarchie bekannte -- Container-Struktur.

In Containern werden die verschiedenen graphischen AWT-Primitive zusammengefaßt; Container können hierarchisch strukturiert sein, d.h. weitere Container enthalten. Der oberste Container einer Applikation bildet gleichsam die sammelnde Ordnungsstruktur. Die AWT verfügt über vier vordefinierte Containertypen:

- [Dialog](#)
Leichtgewichtiges Fenster, auch *Pop-Up-Fenster*; zumeist für Meldungen oder Eingaben benutzt. Mit der Klasse [FileDialog](#) ist ein Standard-Dialog zur Erfragung eines Dateinamens realisiert.
- [Frame](#)
Vergleichbar mit dem gleichnamigen (X)HTML-Konstrukt ermöglichen es Frames eigenständige Fenster innerhalb einer Applikation zu organisieren.
- [Panel](#)
Containertyp, der kein eigenes Fenster öffnet.
- [Window](#)
Container zur Aufnahme von Frames.

Anmerkung: Rückblickend mag es (anfänglich) verwundern, daß die Graphikwiedergabe in Applets ohne Kenntnis und Anwendung der Containerstruktur möglich war. Dies ist jedoch nur auf den

ersten Blick der Fall. Zunächst stellt der Browser oder Appletviewer das Hauptfenster nebst den Menüleisten zur Verfügung -- insofern existiert ein Container zur Aufnahme des Applets bereits durch seine Ausführungsumgebung. Zusätzlich stellt das Applet selbst, durch seine Platzierung in der Vererbungshierarchie unterhalb der Klasse `java.awt.Panel`, einen eigenständigen Container dar.

Layout-Manager

Ein weiterer Ausweis der Zielsetzung plattformunabhängigen Designs des AWT ist die verwirklichte Philosophie zur Platzierung der graphischen Elemente am Bildschirm. Während plattformspezifische GUI-Bibliotheken zumeist die angebotenen Primitive mit absoluten Koordinaten im Anwendungsfenster verankern verfolgt Java hier einen gegensätzlichen Weg.

Durch den Programmierer wird ausschließlich die Platzierung der Komponente im Verhältnis zu anderen Komponenten angegeben, die endgültige Ausrichtung am Schirm nimmt eine zusätzliche AWT-Einheit -- der *Layout-Manager* -- vor. Die Orientierung der Primitiven innerhalb einer Komponente erfolgt weiterhin durch absolute Positionierung.

Im AWT stehen folgende vordefinierte Layout-Manager zur Verfügung:

- Flow -- [FlowLayout](#)
Ordnet die enthaltenen Komponenten zentriert von links nach rechts an.
Eignet sich besonders gut zur Ausrichtung von Button-Leisten.
- Border -- [BorderLayout](#)
Orientiert Komponenten in fünf verschiedene Bereiche.
- Grid -- [GridLayout](#)
Plaziert jedes Element an einer Position innerhalb eines Quadratgitters freiwählbarer Größe.
- Card -- [CardLayout](#)
Organisiert enthaltene Container als Stapel von Karten, die jeweils einzeln angezeigt werden können.
- GridBag -- [GridBag](#)
Richtet Komponenten verschiedener Dimensionen horizontal und vertikal aus.

Anmerkungen:

- Wird die Methode [setLayout\(LayoutManager\)](#) mit dem Übergabeparameter null aufgerufen, so wird kein Layoutmanager verwendet, sondern die Positionierung der graphischen Elemente vollständig dem Anwender überlassen.
- In realen Anwendungen wird üblicherweise massiver Gebrauch von der Möglichkeit der Schachtelung von Layoutmanagern gemacht.
Hierfür wird der Inhalt eines Frames durch mehrere Panels definiert, die unterschiedliche Layoutmanager nutzen.
- Durch Implementierung der Schnittstelle [LayoutManager](#) lassen sich eigene Layout-Manager erzeugen.

Ein erstes Fenster

```
(1)import java.awt.Frame;
(2)
(3)public class AWTEX1 {
(4)    public static void main(String[] args) {
(5)        Frame wnd = new Frame("Einfaches Fenster");
(6)
(7)        wnd.setSize(400,300);
(8)        wnd.setVisible(true);
(9)    } //end main()
(10)} //class AWTEX1
```

Beispiel 91: Ein einfaches Fenster mit AWT [AWTEX1.java](#)

Bildschirmausgabe unter MS-Windows

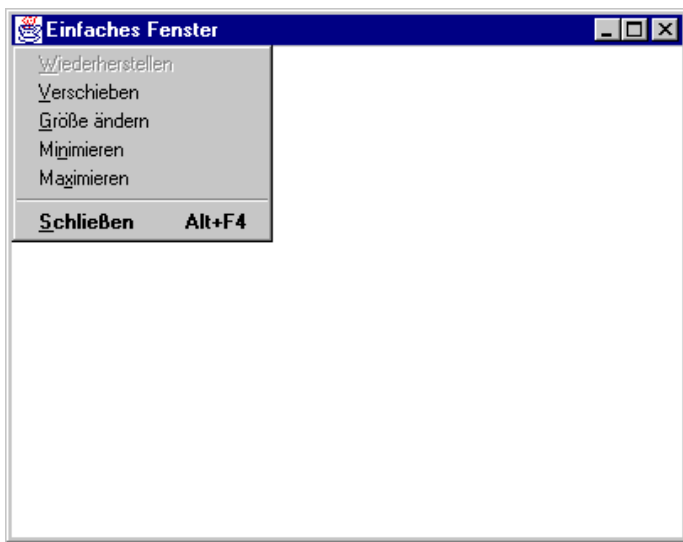


Abbildung 61: Ein einfaches Fenster mit AWT (click on image to enlarge!)

Das Beispiel erzeugt einen Frame mit dem Titel *Einfaches Fenster*.

Durch die, von Component ererbte, Methode [setSize\(int, int\)](#) wird dem leeren Frame die Größe von 400 mal 300 Pixeln zugewiesen.

Zum Abschluß muß der Anzeigevorgang per [setVisible\(boolean\)](#) (ebenfalls von Component ererbt) explizit angestoßen werden.

Die Beiden Methoden `setSize` und `setVisible` greifen auf die visuellen Eigenschaften der Komponente zu, diese und weitere Methoden zur Modifikation der graphischen Erscheinung stehen auf allen Subklassen von Component zur Verfügung.

- [setSize\(Dimension\)](#) und [setSize\(int, int\)](#)
Ändert Größe der Komponente.
- [setVisible\(boolean\)](#)
Zeigt Komponente an, oder verbirgt sie.
- [setLocation\(int, int\)](#)
Verschiebt Komponente an angegeben Position.
- [setBounds\(int, int, int, int\)](#)
Kombination aus `setSize` und `setLocation`, ändert Lage und Größe der Komponente.
- [requestFocus\(\)](#)
Beantragt Eingabefokus.
- [transferFocus\(\)](#)
Gibt Eingabefocus an nächste Komponente weiter.
- [setEnabled\(boolean\)](#)
Aktiviert oder deaktiviert Komponente. Nur aktivierte Komponenten reagieren auf Eingaben und können Ereignisse verarbeiten. Vorgabegemäß sind alle Komponenten aktiviert.
- [setFont\(Font\)](#)
Setzt Zeichensatz der Komponente.
- [setComponentOrientation\(ComponentOrientation\)](#)
Setzt sprach- und kulturspezifische Orientierung der Komponente. Nach dem übergebenen Orientierungsschema werden alle Komponentenelemente und Texte ausgerichtet. Zur Verfügung stehen derzeit `LEFT_TO_RIGHT` und `RIGHT_TO_LEFT`
- [setCursor\(Cursor\)](#)
Setzt Mauszeiger-Bild.
- [setBackground\(Color\)](#)
Setzt Hintergrundfarbe der Komponente.
- [setForeground\(Color\)](#)
Setzt Vordergrundfarbe der Komponente.

Auf Ausprägungen der Klasse [Frame](#) sind ferner verfügbar:

- [setState\(int\)](#)
Schaltet zwischen den Zuständen `Frame.NORMAL` (Vorgabe) und `Frame.ICONIFIED` (verkleinert) um.
- [setIconImage\(Image\)](#)
Definiert Icon zur Anzeige bei minimierter Darstellung.
- [setMenuBar\(MenuBar\)](#)
Definiert Pull-down-Menü zur Verwendung innerhalb des Frames.
- [setResizable\(boolean\)](#)
Erlaubt oder Verbietet Größenänderung durch Anwender.
- [setTitle\(String\)](#)

Setzt Titelzeile des Frames.

Bei der Ausführung fällt zunächst die Existenz der plattformüblichen Standardmenüs auf. Sie werden automatisch durch die Ausführungsumgebung zur Verfügung gestellt und mit Funktionalität versehen.

Jedoch zeigt sich, daß es nicht möglich ist das Fenster mit den üblichen Betriebssystem-spezifischen Methoden (*Close-Button*, Auswahl des entsprechenden Menüs, Tastenkombination, etc.) zu schließen. Erst der Abbruch des Prozesses der virtuellen Maschine terminiert die Applikation.

Denn anders als die üblichen Vorgabeaktionen ist das hierfür erforderliche Applikationsverhalten durch den Anwender selbst bereitzustellen.

Ereignisbehandlung

Im Allgemeinen erfolgt die Kommunikation zwischen einer graphischen Oberfläche und dem ausführenden Betriebssystem in Form von Nachrichten, die zwischen diesen beiden Kommunikationspartnern ausgetauscht werden.

Das Betriebssystem unterrichtet die Applikation über den Eintritt bestimmter Ereignisse (engl. *event*), hierzu zählen u.a. Mausbewegungen und -klicks, Tastaturanschläge sowie alle Fensteroperationen.

Am Nachrichtenverkehr sind prinzipiell drei Objekte des Systems beteiligt: Die Ereignisquelle, das Objekt bei dem das Ereignis eintrat (z.B. im Falle eines Mausklicks: der entsprechende Button); der Ereignisempfänger, ein Objekt das auf das eingetretene Ereignis reagiert; die Nachricht selbst, in objektorientierten System als eigenständiges Objekt realisiert, welches das ausgelöste Ereignis näher beschreibt.

Dieser Kommunikationsmechanismus ist als *Delegation Based Event Handling* bekannt, da jedes Ereignis an eine konkrete Instanz delegiert wird, die nachfolgend die Behandlung übernimmt. Es bietet zwei entscheidende Vorteile:

- Verringerung des Nachrichtenverkehrs innerhalb der Applikation, durch klare Entscheidbarkeit des Nachrichtenweges.
Insbesondere unterbleibt das Senden von Nachrichten, für die kein Empfänger existiert.
- Klare Trennung zwischen GUI-Code und Applikationslogik.

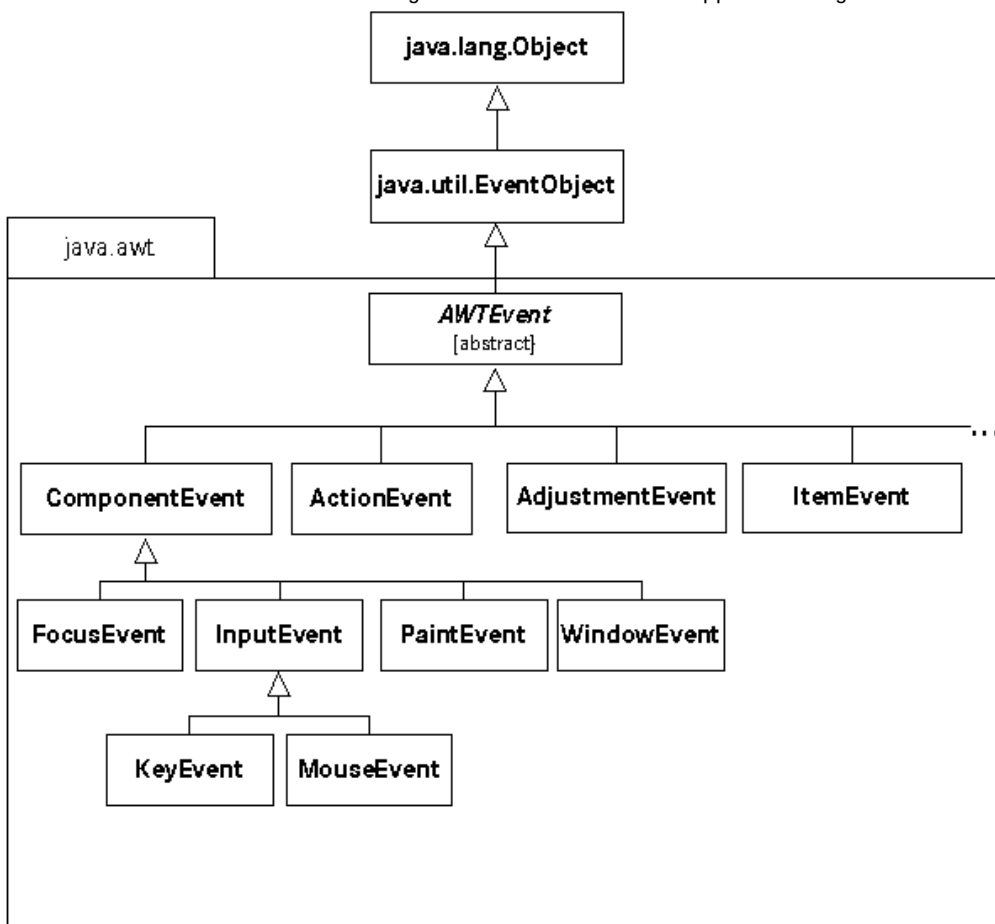


Abbildung 62: Hierarchie der AWT-Ereignisse (click on image to enlarge!)

Die Abbildung zeigt eine Auswahl der verschiedenen Ereignistypen. Wie dargestellt sind alle

spezifischen Ereignisse von der Klasse [EventObject](#) abgeleitet. Neben den AWT-Ereignissen existiert eine Fülle weiterer Ereignistypen, die in den verschiedenen spezifischen Paketen (wie `java.beans`, `javax.sound`, etc.) untergebracht sind.

Durch die Superklasse `EventObject` wird auch die an alle Unterklassen vererbte Methode [getSource\(\)](#) definiert, welche zu jeder `EventObject`-Ausprägung die auslösende Instanz liefert. Unterhalb der Klasse [ComponentEvent](#) sind die *low-level*-Ereignisse eingeordnet, die auf der Ebene einer visuellen Komponente auftreten können.

Die Ereignisse [ActionEvent](#), [ItemEvent](#) und [TextEvent](#) werden als *semantische* Ereignisse bezeichnet, da sie nicht an ein konkretes visuelles Element gebunden sind.

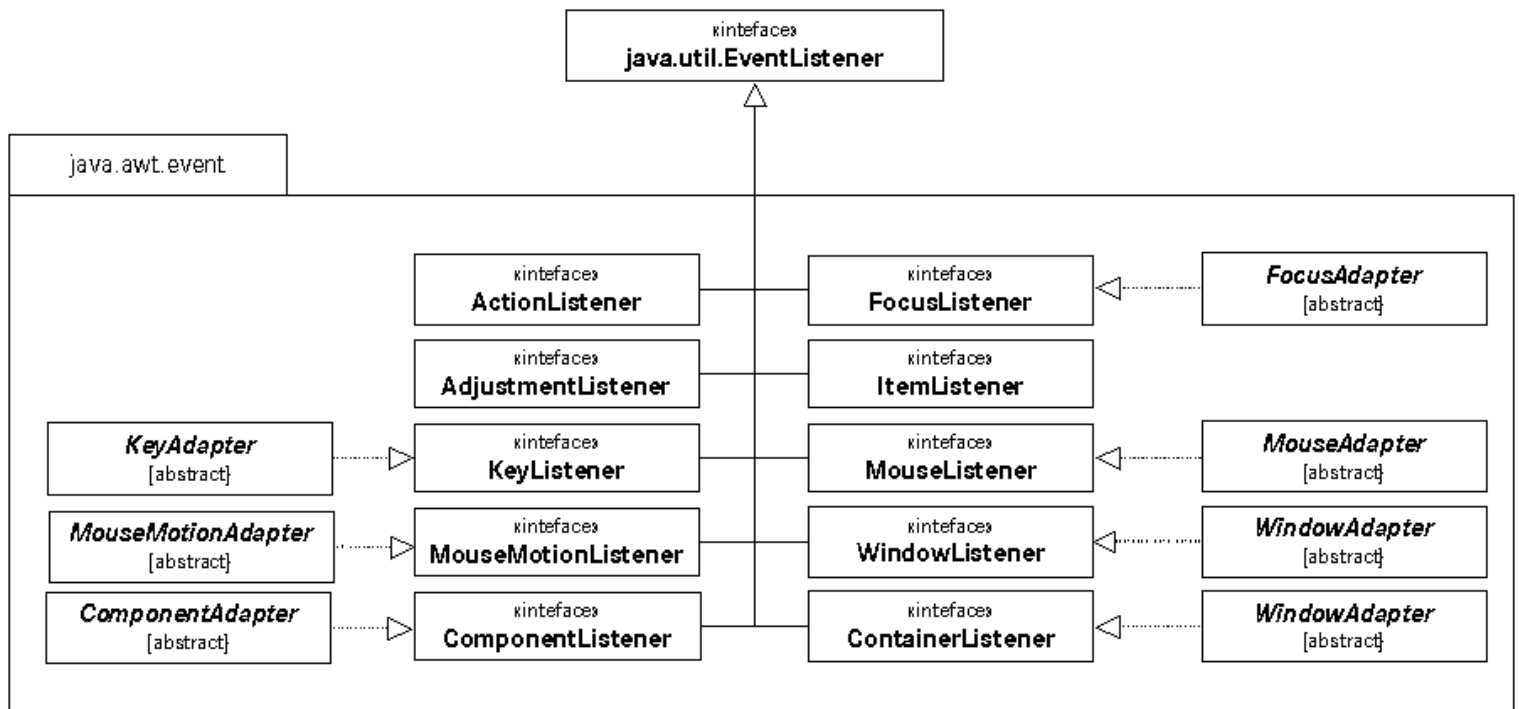


Abbildung 63: Die Event-Listener-Schnittstellen und die zugehörigen Adapter-Klassen (click on image to enlarge!)

Zur Verarbeitung der verschiedenen Nachrichtentypen muß der Empfänger eine Reihe von Methoden implementieren, die durch entsprechende Schnittstellen definiert sind. Die Abbildung stellt eine Auswahl der angebotenen Schnittstellen nebst den bereits angebotenen Implementierungen dar.

Schablone zur Reaktion auf AWT-Ereignisse

Die Reaktion auf AWT-Ereignisse wird durch sog. *event handler* übernommen; diese Objekte müssen die Benachrichtigung im Falle des Auftretens eines Ereignisses explizit anmelden.

Die Schritte im Einzelnen:

1. Definition einer Event-Handler-Klasse durch Implementierung der entsprechenden Schnittstelle
2. Registrierung des Event-Handlers

Somit kann auch das [vorangegangene Beispiel](#) entsprechend erweitert werden, um die noch ausstehende Reaktion auf das *Beenden*-Ereignis zu verwirklichen.

```

(1)import java.awt.Frame;
(2)import java.awt.Window;
(3)import java.awt.event.WindowAdapter;
(4)import java.awt.event.WindowEvent;
(5)
(6)public class AWTEx2 {
(7)    public static void main(String[] args) {
(8)        Frame wnd = new Frame("Einfaches Fenster");
(9)        wnd.addWindowListener( new WindowClosingAdapter() );
(10)        wnd.setSize(400,300);
(11)        wnd.setVisible(true);
(12)    } //main()
(13)} //class AWTEx2
(14)
(15)class WindowClosingAdapter extends WindowAdapter {
(16)    public void windowClosing( WindowEvent we) {
  
```

```

(17)         Window wnd = we.getWindow();
(18)
(19)         wnd.setVisible(false);
(20)         wnd.dispose();
(21)         System.exit(0);
(22)     } //windowClosing()
(23)} //class WindowClosingAdapter

```

Beispiel 92: Ein einfaches Fenster mit AWT [AWTEx2.java](#)

Die Applikation erweitert das bisherige Beispiel um eine Adapter-Klasse, welche auf Fensterereignisse reagiert. Hierzu wird eine Ausprägung der Adapter-Klasse `WindowClosingAdapter` als Window Listener mit der Methode `addWindowListener` (`WindowListener`) registriert.

Die Adapter-Klasse erweitert die bestehende API-Klasse `WindowAdapter` und implementiert damit die Schnittstelle `WindowListener`. Sie definiert die überschriebene Methode `windowClosing` (`WindowEvent`), die bei auftreten des Ereignisses automatisch zur Verarbeitung aufgerufen wird. Nach Ermittlung des Ereignis-sendenden Fensters per `getWindow()` wird dieses Fenster zunächst ausgeblendet (`setVisible(false)`) und im Anschluß zerstört (`dispose()`). Danach terminiert die Applikation die virtuelle Maschine.

```

(1)import java.awt.Button;
(2)import java.awt.FlowLayout;
(3)import java.awt.Frame;
(4)import java.awt.Label;
(5)import java.awt.Window;
(6)import java.awt.event.MouseAdapter;
(7)import java.awt.event.MouseEvent;
(8)import java.awt.event.MouseEvent;
(9)import java.awt.event.MouseMotionAdapter;
(10)import java.awt.event.WindowAdapter;
(11)import java.awt.event.WindowEvent;
(12)
(13)public class AWTEx3 {
(14)    private Label mousePos,
(15)                                     counterVal;
(16)
(17)    public static void main(String[] args) {
(18)        AWTEx3 app = new AWTEx3();
(19)    } //main()
(20)
(21)    public AWTEx3() {
(22)        Frame wnd = new Frame("Einfaches Fenster");
(23)        mousePos = new Label("mouse pos");
(24)        counterVal = new Label("0");
(25)        Button counter = new Button("add one!");
(26)
(27)        wnd.setSize(200,100);
(28)        wnd.setLayout(new FlowLayout() );
(29)        wnd.add (mousePos);
(30)        wnd.add(counter);
(31)        wnd.add(counterVal);
(32)
(33)        wnd.addWindowListener( new WindowClosingAdapter() );
(34)        wnd.addMouseMotionListener( new MyMouseMotionAdapter(this) );
(35)        counter.addMouseListener( new CounterClicked(this) );
(36)        wnd.addMouseListener( new WindowClicked(this) );
(37)
(38)        wnd.setVisible(true);
(39)    } //constructor
(40)
(41)    public void setMousePos(String newText) {
(42)        mousePos.setText( newText );
(43)    } //setMousePos()
(44)
(45)    public void incrementCounter() {
(46)        counterVal.setText( ""+(Integer.parseInt(counterVal.getText()+1) );
(47)    } //incrementCounter()
(48)    public void decrementCounter() {
(49)        counterVal.setText( ""+(Integer.parseInt(counterVal.getText()-1) );

```

```

(50)    } //decrementCounter()
(51)} //class AWTEX3
(52)
(53)class WindowClosingAdapter extends WindowAdapter {
(54)    public void windowClosing(WindowEvent we) {
(55)        Window wnd = we.getWindow();
(56)
(57)        wnd.setVisible(false);
(58)        wnd.dispose();
(59)        System.exit(0);
(60)    } //windowClosing()
(61)} //class WindowClosingAdapter
(62)
(63)class CounterClicked extends MouseAdapter {
(64)    private AWTEX3 app;
(65)
(66)    public CounterClicked(AWTEX3 app) {
(67)        this.app = app;
(68)    } //constructor
(69)
(70)    public void mouseClicked( MouseEvent me) {
(71)        app.incrementCounter();
(72)    } //mouseClicked()
(73)} //class CounterClicked
(74)
(75)class WindowClicked extends MouseAdapter {
(76)    private AWTEX3 app;
(77)
(78)    public WindowClicked(AWTEX3 app) {
(79)        this.app = app;
(80)    } //constructor
(81)
(82)    public void mouseClicked( MouseEvent me) {
(83)        app.decrementCounter();
(84)    } //mouseClicked()
(85)} //class WindowClicked
(86)
(87)class MyMouseMotionAdapter extends MouseMotionAdapter {
(88)    private AWTEX3 app;
(89)
(90)    public MyMouseMotionAdapter(AWTEX3 app) {
(91)        this.app = app;
(92)    } //constructor
(93)
(94)    public void mouseMoved( MouseEvent me) {
(95)        app.setMousePos( "("+me.getX()+", "+me.getY()+")" );
(96)    } //mouseMoved()
(97)} //class myMouseAdapter

```

Beispiel 93: Event-Handling [AWTEX3.java](#)

Bildschirmausgabe



Abbildung 64: Zwischenzustand der Applikation (click on image to enlarge!)

Die Applikation definiert, unter Verwendung des Flow-Layouts, zwei Beschriftungsfelder (labels) mousePos und counterVal sowie eine mit add one! beschriftete Schaltfläche.

Anmerkung: Die Verwendung des Layoutmanagers ist zur Verarbeitung der Maus-Events zwingend notwendig!

Vor Anzeige des Applikationsfensters werden vier Ereignis-Handler registriert: Der bekannte WindowClosingListener zur Umsetzung des *Schließen*-Ereignisses, der MouseMotionListener zur Verfolgung der Mausbewegungen im Hauptfenster sowie zwei MouseListener die auf Mausereignisse -- außer Bewegungen -- auf der definierten Schaltfläche und dem Hauptfenster selbst reagieren.

Die Umsetzung des WindowClosingListeners ist gegenüber [dem vorhergehenden Beispiel](#) unverändert.

Innerhalb der Klassen counterClicked und windowClicked wird jeweils die Methode mouseClicked der Schnittstelle [MouseListener](#) überschrieben, sie wurde von der Superklasse MouseAdapter geerbt. Wird ein Mausklick auf den Button registriert, so wird eine Methode der Hauptklasse awtEx3 aufgerufen, welche die Beschriftung des Textfeldes entsprechend verändert. Beim Mausklick ist es in der vorliegenden Implementierung unentscheidend, mit welcher Maustaste dieser erfolgte. Eine nähere Analyse, beispielsweise hinsichtlich der gedrückten Taste, des aufgetretenen Ereignisses kann über [Attribute der Klasse MouseEvent](#) erfolgen.

Die Spezialisierung der Klasse [MouseMotionAdapter](#) realisiert die Methode [mouseMoved \(MouseEvent\)](#). Sie wird jeweils bei Änderung der Mauskoordinate aufgerufen. Die beiden Koordinaten werden innerhalb des Methodenkörpers mittels der angebotenen API-Methoden extrahiert und an die Hauptklasse zur Anzeige übergeben.

Event Handling und anonyme innere Klassen

Bei näherer Betrachtung der verschiedenen Event-Handler des vorhergehenden Beispiels fällt deren immergleiche Grundstruktur auf: Zunächst das Erben von einer passenden Adapterklasse um eine oder mehrere Methoden zu überschreiben. Im konkreten Beispiel wird sogar zweimal dieselbe Adapterklasse überschrieben; jedoch mit unterschiedlichem Verhalten.

Die durch die Spezialisierung entstehenden Klassen werden in allen Fällen zur Erzeugung genau eines Objekts -- des jeweiligen Listeners -- herangezogen; eine sonstige Weiter- oder Wiederverwendung in der Applikation liegt nicht vor, und ist unter Berücksichtigung des Klassendesigns auch für die Zukunft nicht anzunehmen.

Nimmt man die beiden Randbedingungen -- *Vererbung und Überschreiben ererbter Methoden* und *keine Wiederverwendung* -- zusammen, so offenbart sich das Sprachmittel der [anonymen inneren Klassen](#) als wesentlich adäquater zur Lösung der vorliegenden Problemstellung.

Unter Nutzung dieses Mechanismus ergibt läßt sich der Code des Beispiels wie folgt modifizieren:

```
(1)import java.awt.Button;
(2)import java.awt.FlowLayout;
(3)import java.awt.Frame;
(4)import java.awt.Label;
(5)import java.awt.Window;
(6)import java.awt.event.MouseAdapter;
(7)import java.awt.event.MouseEvent;
(8)import java.awt.event.MouseMotionAdapter;
(9)import java.awt.event.WindowAdapter;
(10)import java.awt.event.WindowEvent;
(11)
(12)public class AWTEEx3i {
(13)    private Label mousePos,
(14)                                counterVal;
(15)
(16)    private static AWTEEx3i app;
(17)
(18)    public static void main(String[] args) {
(19)        app = new AWTEEx3i();
(20)    } //main()
(21)
(22)    public AWTEEx3i() {
(23)        Frame wnd = new Frame("Einfaches Fenster");
(24)        mousePos = new Label("mouse pos");
(25)        counterVal = new Label("0");
(26)        Button counter = new Button("add one!");
(27)
(28)        wnd.setSize(200,100);
(29)        wnd.setLayout(new FlowLayout() );
(30)        wnd.add (mousePos);
(31)        wnd.add(counter);
(32)        wnd.add(counterVal);
(33)
(34)        wnd.addWindowListener( new WindowAdapter() {
(35)            public void windowClosing(WindowEvent we) {
(36)                Window wnd = we.getWindow();
(37)
(38)                wnd.setVisible(false);
(39)                wnd.dispose();
```

```

(40)                System.exit(0);
(41)                } //windowClosing()
(42)            } //anonymous inner class
(43)        );
(44)
(45)        wnd.addMouseMotionListener( new MouseMotionAdapter() {
(46)            public void mouseMoved( MouseEvent me) {
(47)                app.setMousePos( "("+me.getX()+","+me.getY()+")" );
(48)            } //mouseMoved()
(49)        } //anonymous inner class
(50)    );
(51)
(52)        counter.addMouseListener( new MouseAdapter() {
(53)            public void mouseClicked( MouseEvent me)      {
(54)                app.incrementCounter();
(55)            } //mouseClicked()
(56)        } //anonymous inner class
(57)    );
(58)
(59)        wnd.addMouseListener( new MouseAdapter() {
(60)            public void mouseClicked( MouseEvent me) {
(61)                app.decrementCounter();
(62)            } //mouseClicked()
(63)        } //anonymous inner class
(64)    );
(65)
(66)        wnd.setVisible(true);
(67)    } //constructor
(68)
(69)    public void setMousePos(String newText) {
(70)        mousePos.setText( newText );
(71)    } //setMousePos()
(72)
(73)    public void incrementCounter() {
(74)        counterVal.setText( ""+(Integer.parseInt(counterVal.getText())+1) );
(75)    } //incrementCounter()
(76)
(77)    public void decrementCounter() {
(78)        counterVal.setText( ""+(Integer.parseInt(counterVal.getText())-1) );
(79)    } //decrementCounter()
(80)} //class AWTE3i

```

Beispiel 94: Das Beispiel AWTE3 unter Nutzung anonymer innerer Klassen [AWTE3i.java](#)

Der entstehende Code wird deutlich kompakter, und lokalisiert zusätzlich die Handlermethoden in den Registrierungsaufrufen.

Da anonyme innere Klassen (naturgemäß) keine Konstruktoren besitzen können wurden die Methoden entsprechend modifiziert.

Event-Handling im Überblick

Schnittstelle	Methoden	Parameter / Zugriffsmethoden Eventauslöser	
ActionListener	actionPerformed(ActionEvent)	ActionEvent getActionCommand() getModifiers()	Button List MenuItem TextField
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)	AdjustmentEvent getAdjustable() getAdjustmentType() getValue()	Scrollbar
ItemListener	itemStateChanged(ItemEvent)	ItemEvent getItem() getItemSelectable() getStateChange()	Checkbox CheckboxMenuItem Choice List

<u>TextListener</u>	<u>textValueChanged(TextEvent)</u>	<u>TextEvent</u>	<u>TextComponent</u>
<u>ComponentListener</u>	<u>componentHidden(ComponentEvent)</u> <u>componentMoved(ComponentEvent)</u> <u>componentResized(ComponentEvent)</u> <u>componentShown(ComponentEvent)</u>	<u>ComponentEvent</u> <u>getComponent()</u>	<u>Component</u>
<u>ContainerListener</u>	<u>componentAdded(ContainerEvent)</u> <u>componentRemoved(ContainerEvent)</u>	<u>ContainerEvent</u> <u>getChild()</u> <u>getContainer()</u>	<u>Container</u>
<u>FocusListener</u>	<u>focusGained(FocusEvent)</u> <u>focusLost(FocusEvent)</u>	<u>FocusEvent</u> <u>isTemporary()</u>	<u>Component</u>
<u>KeyListener</u>	<u>keyPressed(KeyEvent)</u> <u>keyReleased(KeyEvent)</u> <u>keyTyped(KeyEvent)</u>	<u>KeyEvent</u> <u>getKeyChar()</u> <u>getKeyCode()</u> <u>getKeyModifiersText(int)</u> <u>getKeyText()</u> <u>isActionKey()</u>	<u>Component</u>
<u>MouseListener</u>	<u>mouseClicked(MouseEvent)</u> <u>mouseEntered(MouseEvent)</u> <u>mouseExited(MouseEvent)</u> <u>mousePressed(MouseEvent)</u> <u>mouseReleased(MouseEvent)</u>	<u>MouseEvent</u> <u>getClickCount()</u> <u>getPoint()</u> <u>getX()</u> <u>getY()</u> <u>isPopupTrigger()</u>	<u>Component</u>
<u>MouseMotionListener</u>	<u>mouseDragged(MouseEvent)</u> <u>mouseMoved(MouseEvent)</u>	<u>MouseEvent</u> <u>getClickCount()</u> <u>getPoint()</u> <u>getX()</u> <u>getY()</u> <u>isPopupTrigger()</u>	<u>Component</u>
<u>WindowListener</u>	<u>windowActivated(WindowEvent)</u> <u>windowClosed(WindowEvent)</u> <u>windowClosing(WindowEvent)</u> <u>windowDeactivated(WindowEvent)</u> <u>windowDeiconified(WindowEvent)</u> <u>windowIconified(WindowEvent)</u> <u>windowOpened(WindowEvent)</u>	<u>WindowEvent</u> <u>getWindow()</u>	<u>Window</u>

Menüs

Das AWT bietet die auch von anderen graphischen Benutzerschnittstellen bekannten Menüprimitive und -Grundfunktionen an. Darunter verschachtelte Menüs (Submenüs), Short-Cuts, frei platzierbare Kontextmenüs, etc.

Die Abbildung zeigt die verschiedenen Primitive und ihre Beziehungen zueinander.

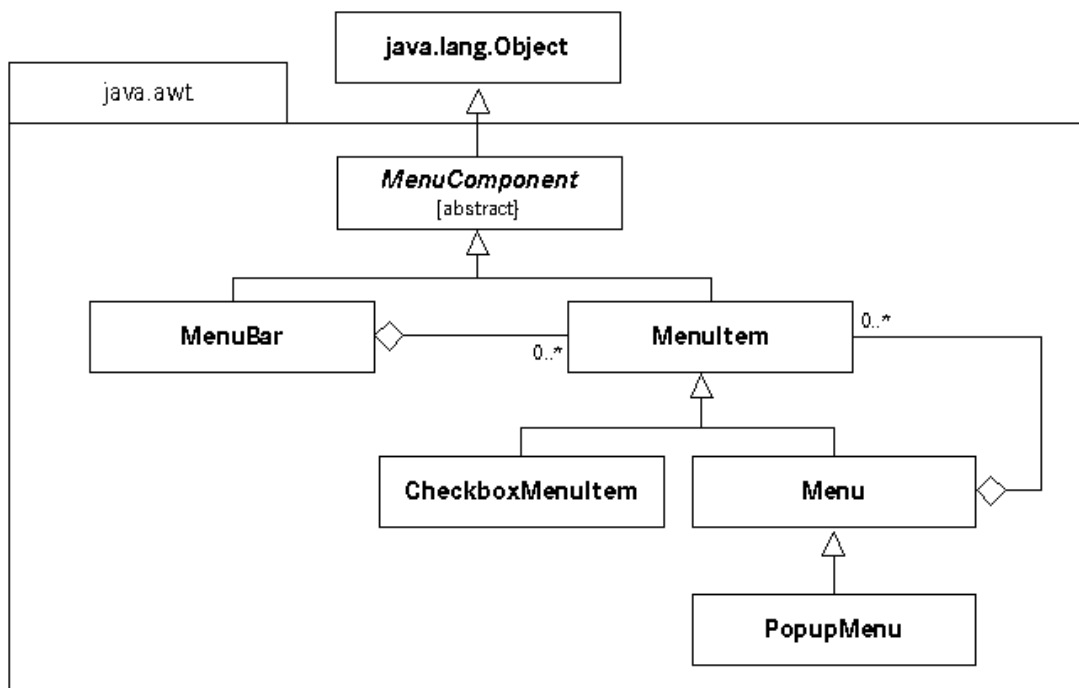


Abbildung 65: Struktur der Menüprimitive der AWT (click on image to enlarge!)

Die abstrakte Klasse [MenuComponent](#) bildet die Wurzel der Menüklassenhierarchie.

Ein [MenuBar](#) stellt eine Zusammenfassung von Menüs dar. Objekte dieser Klasse bilden den Ausgangspunkt einer Menüstruktur.

Innerhalb eines MenuBars können beliebig viele [MenuItem](#)-Ausprägungen organisiert werden. Ein MenuItem ist ein beliebiger Eintrag eines Menüs. Hierbei kann es sich um einen anklickbaren Menüeintrag oder ein weiteres Menü handeln, welches sich bei Mausberührung öffnet.

Üblicherweise werden konkrete Menüeinträge in Menüs zusammengefaßt. Dem Menü selbst entspricht die AWT-Klasse [Menu](#). Jedes Menu-Objekt kann aus weiteren MenuItems bestehen. Mithin kann ein Menü Submenüs oder direkte Menüeinträge in beliebiger Reihenfolge enthalten.

Als Spezialisierung der durch die Klasse MenuItem definierten Menüeinträge stehen umschaltbare Menüpunkte durch die Klasse [CheckboxMenuItem](#) zur Verfügung. Gegenüber den herkömmlichen Menüeinträgen verfügt dieser Typ über ein zusätzliches -- in der visuellen Darstellung plattformabhängig variierendes -- Symbol welches den Eintragszustand „aktiviert“ oder „deaktiviert“ anzeigt.

Menüleisten-unabhängige Kontextmenüs werden durch die Klasse [PopupMenu](#) realisiert. Sie sind frei platzierbar, verhalten sich jedoch ansonsten wie die bekannten Menüs.

Neue (klickbare) Menüeinträge werden als Objekte der Klasse [MenuItem](#) erzeugt. Die dem Konstruktor übergebene Zeichenkette legt den späteren angezeigten Eintrag im Menü fest. ([Verwendung im Beispiel](#)).

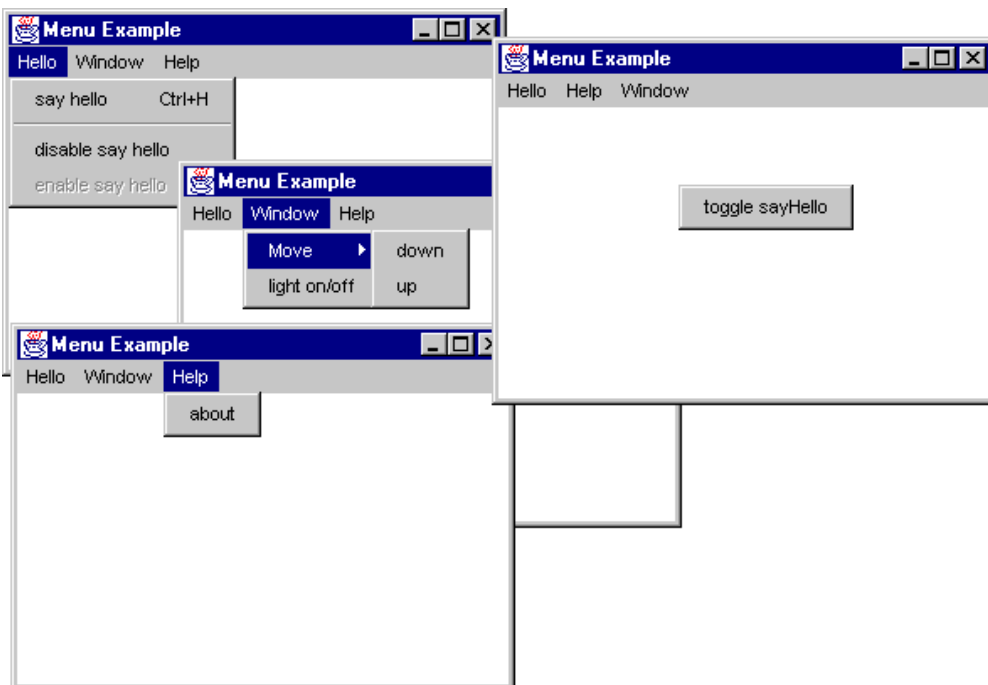


Abbildung 66: Menüs der Beispielapplikation (click on image to enlarge!)

Die Abbildung stellt einen Ausschnitt der Menüstruktur der Beispielapplikation dar:

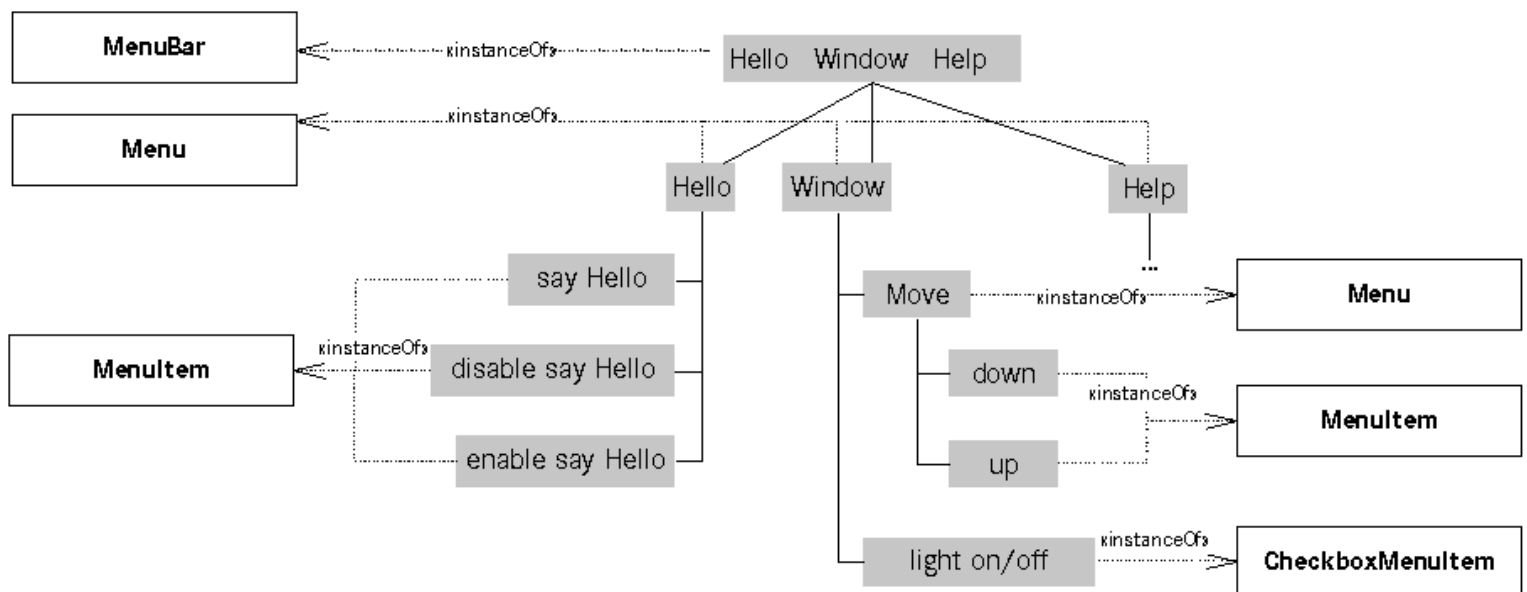


Abbildung 67: Menüstruktur der Beispielapplikation (click on image to enlarge!)

Erzeugung von Menüstrukturen

Ausgangspunkt der Menüerstellung ist immer ein Objekt der Klasse [MenuBar](#). Durch den parameterlosen Konstruktor wird eine leere Menüleiste erzeugt, die als Container zur Aufnahme der weiteren Menükomponenten dient. ([Verwendung im Beispiel](#)).

Die einzelnen Menüs werden durch Objekte der entsprechenden Klasse [Menu](#) repräsentiert. Der in der Menüleiste aufgeführte Menüname wird als Zeichenkettenparameter (Klasse `java.lang.String`) übergeben. ([Verwendung im Beispiel](#).)

Durch die Methode [add\(MenuItem\)](#) werden Menüeinträge oder ganze Menüs -- allgemein: Subklassen von [MenuItem](#) -- in eine bestehende Menüleiste eingehängt. ([Verwendung im Beispiel](#)).

Zusätzlich kann dem Konstruktor eine Instanz der Klasse [MenuShortcut](#) übergeben werden. Hierdurch wird die in GUIs übliche direkte Anwahlmöglichkeit einzelner Menüeinträge über Tastenkürzel realisiert. ([Verwendung im Beispiel](#)).

Zur Erzeugung von [CheckboxMenuItems](#), Menüeinträgen die ihren zweiwertigen Zustand automatisch darstellen, muß ein Objekt der Klasse [CheckboxMenuItem](#) erzeugt und in ein bestehendes Menü eingehängt werden. ([Verwendung im Beispiel](#)).

Die Variante der Pop-up Menüs wird durch die gleichnamige Klasse [PopupMenu](#) erzeugt. Im

Unterschied zu den herkömmlichen Menüs wird dieser Menütyp nicht in der Menüleiste, sondern dem beherrschenden Fenster direkt, durch die dortige add-Methode, eingehängt. ([Verwendung im Beispiel](#)).

Die **Ereignisbehandlung** für Menüstrukturen folgt dem in der Übersicht [Event-Handling im Überblick](#) gegebenen Schema.

Für jeden Menüeintrag vom Typ [MenuItem](#) muß ein Objekt vom Typ [ActionListener](#) über die Methode [addActionListener\(ActionListener\)](#) der Klasse [MenuItem](#) registriert werden. Dieser muß die Methode [actionPerformed\(ActionEvent\)](#) überschreiben. Dort findet sich der Code, welcher bei Eintreten des Ereignisses (Mausclick auf Menüeintrag) abgearbeitet wird. ([Verwendung im Beispiel](#)).

Menüpunkte der Klasse [CheckboxMenuItem](#) werden durch Objekte die die Schnittstelle [ItemListener](#) implementieren überwacht. Tritt die Zustandsänderung ein, so wird die Methode [itemStateChanged\(ItemEvent\)](#) aufgerufen. ([Verwendung im Beispiel](#)).

```
(1)import java.awt.Button; (2)import java.awt.CheckboxMenuItem; (3)import java.awt.Color; (4)import java.
  awt.Dialog; (5)import java.awt.FlowLayout; (6)import java.awt.Frame; (7)import java.awt.Label; (8)import
  java.awt.Menu; (9)import java.awt.MenuBar; (10)import java.awt.MenuItem; (11)import java.awt.MenuShortcut;
  (12)import java.awt.Point; (13)import java.awt.PopupMenu; (14)import java.awt.Window; (15)import java.awt.
  event.ActionEvent; (16)import java.awt.event.ActionListener; (17)import java.awt.event.InputEvent; (18)
  import java.awt.event.ItemEvent; (19)import java.awt.event.ItemListener; (20)import java.awt.event.
  KeyEvent; (21)import java.awt.event.MouseAdapter; (22)import java.awt.event.MouseEvent; (23)import java.awt.
  event.WindowAdapter; (24)import java.awt.event.WindowEvent; (25) (26)public class AWTE4 { (27) private
  static AWTE4 app; (28) private Label lbl_hello; (29) private Frame wnd; (30) private MenuItem mi_sayHello,
  mi_disable, mi_enable, mi_toggle, mi_up, mi_down; (31) private CheckboxMenuItem mi_light; (32) private
  PopupMenu myPopupMenu; (33) (34) class ToggleSayHelloState implements ActionListener { (35) public void
  actionPerformed(ActionEvent ae) { (36) mi_sayHello.setEnabled( !mi_sayHello.isEnabled() ); (37) mi_enable.
  setEnabled( !mi_sayHello.isEnabled() ); (38) mi_disable.setEnabled( mi_sayHello.isEnabled() ); (39) } //
  actionPerformed() (40) } //class ToggleSayHelloState (41) (42) public static void main(String[] args) { (43)
  app = new AWTE4(); (44) } //main() (45) (46) public AWTE4() { (47)wnd = new Frame("Menu Example"); (48)
  lbl_hello = new Label("Hello!"); (49) wnd.add(lbl_hello); (50) (51) wnd.setSize(300,200); (52) wnd.setLayout
  (new FlowLayout() ); (53) (54) MenuBar myMenuBar = new MenuBar(); (55) Menu m_hello = new Menu("Hello");
  (56) myMenuBar.add(m_hello); (57) mi_sayHello = new MenuItem("say hello", new MenuShortcut (KeyEvent.
  VK_H)); (58) m_hello.add( mi_sayHello ); (59) mi_disable = new MenuItem("disable say hello"); (60) m_hello.
  addSeparator(); (61) mi_enable = new MenuItem("enable say hello"); (62) m_hello.add( mi_disable ); (63)
  m_hello.add( mi_enable ); (64) (65) Menu m_windowAction = new Menu("Window"); (66) Menu m_windowActionMove
  = new Menu("Move"); (67) mi_light = new CheckboxMenuItem("light on/off"); (68) mi_down = new MenuItem
  ("down"); (69) mi_up = new MenuItem ("up"); (70) (71) m_windowActionMove.add(mi_down); (72)
  m_windowActionMove.add(mi_up); (73) m_windowAction.add(m_windowActionMove); (74) m_windowAction.add
  (mi_light); (75) myMenuBar.add(m_windowAction); (76) (77) //pop-up menu (78) myPopupMenu = new PopupMenu();
  (79) MenuItem mi_toggle = new MenuItem("toggle sayHello"); (80) myPopupMenu.add( mi_toggle ); (81) wnd.add
  ( myPopupMenu ); (82) (83) Menu m_hlp = new Menu("Help"); (84) m_hlp.add(new MenuItem("about")); (85) (86)
  myMenuBar.setHelpMenu(m_hlp); (87) (88) wnd.setMenuBar(myMenuBar); (89) (90) mi_up.addActionListener( new
  ActionListener() { (91) public void actionPerformed(ActionEvent ae) { (92) Point pnt = wnd.getLocation();
  (93) (94) for (int x=pnt.x; x >= 0 ; x--) (95) wnd.setLocation(x, pnt.y); (96) for (int y=pnt.y; y >= 0;
  y--) (97) wnd.setLocation(0, y); (98) } //actionPerformed() (99) } //anonymous inner class (100) ); (101)
  (102) mi_down.addActionListener( new ActionListener() { (103) public void actionPerformed(ActionEvent ae)
  { (104) final Dialog dlg_notImplemented = new Dialog (wnd,true); (105) Label lbl = new Label("function not
  implemented, yet!"); (106) dlg_notImplemented.setTitle("Sorry, I'm afraid I can't to that ..."); (107)
  dlg_notImplemented.setLayout( new FlowLayout() ); (108) dlg_notImplemented.setResizable(false); (109)
  dlg_notImplemented.add(lbl); (110) dlg_notImplemented.setSize(250,80); (111) (112) Button btn_ok = new
  Button("ok"); (113) btn_ok.addActionListener( new ActionListener() { (114) public void actionPerformed
  (ActionEvent ae) { (115) dlg_notImplemented.setVisible(false); (116) dlg_notImplemented.dispose();
  (117) } //actionPerformed() (118) } //anonymous inner class (119) ); (120) (121) dlg_notImplemented.add
  (btn_ok); (122) (123) dlg_notImplemented.show(); (124) } //actionPerformed() (125) } //anonymous inner
  class (126) ); (127) (128) mi_light.addItemListener( new ItemListener() { (129) public void itemStateChanged
  (ItemEvent ie) { (130) if (mi_light.getState()) (131) wnd.setBackground( new Color(33,33,33) ); (132) else
  (133) wnd.setBackground( new Color(255,255,255) ); (134) } //actionPerformed() (135) } //anonymous inner
  class (136) ); (137) (138) mi_sayHello.addActionListener( new ActionListener() { (139) public void
  actionPerformed(ActionEvent ae) { (140) lbl_hello.setVisible(true); (141) try { (142) Thread.sleep(1500);
  (143) } catch (InterruptedException e) { (144) //ignore it (145) } //catch (146) lbl_hello.setVisible
  (false); (147) } //actionPerformed() (148) } //anonymous inner class (149) ); (150) (151) ActionListener
  toggler = new ToggleSayHelloState(); (152) mi_disable.addActionListener( toggler ); (153) mi_enable.
  addActionListener( toggler ); (154) mi_toggle.addActionListener( toggler ); (155) (156) wnd.
  addWindowListener( new WindowAdapter() { (157) public void windowClosing(WindowEvent we) { (158) Window wnd
  = we.getWindow(); (159) (160) wnd.setVisible(false); (161) wnd.dispose(); (162) System.exit(0); (163) } //
  windowClosing() (164) } //anonymous inner class (165) ); (166) (167) wnd.addMouseListener( new MouseAdapter
  () { (168) public void mouseClicked( MouseEvent me) { (169) if ( ( me.getModifiers() & InputEvent.
  BUTTON1_MASK) == 0 ) (170) myPopupMenu.show( me.getComponent(), me.getX(), me.getY() ); (171) } //
  mouseClicked() (172) } //anonymous inner class (173) ); (174) (175) wnd.setVisible(true); (176) lbl_hello.
  setVisible(false); (177) mi_enable.setEnabled(false); (178) } //main() (179)} //class AWTE4
```

Beispiel 95: Beispiel einer Menüstruktur [AWTE4.java](#)

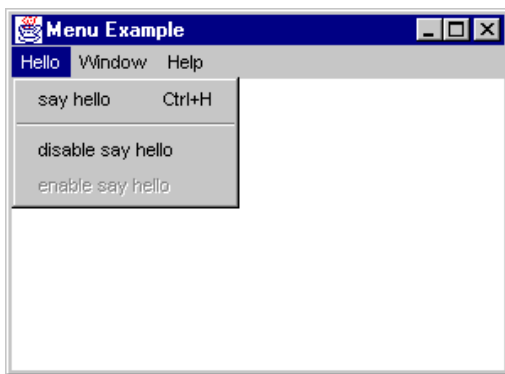


Abbildung 68: Das Menü Hello (click on image to enlarge!)

Das Menü *Hello* der Beispielapplikation besteht aus drei Menüeinträgen -- *say hello*, *enable say hello* und *disable say hello*. Für den ersten Menüpunkt ist das Tastaturkürzel CTRL+H definiert ([Definition im Code](#)). Bereits zum Startzeitpunkt der Applikation ist der Menüpunkt *enable say hello* deaktiviert, da der entsprechende Menüpunkt vorgabegemäß aktiviert ist. Die Instanzenmethode [setEnabled](#) der Klasse `MenuItem` erlaubt den Wechsel zwischen den beiden Zuständen „aktiviert“, mit normaler visueller Darstellung des Menüpunktes, und „deaktiviert“, mit entsprechender *ausgegrauter* Darstellung. Standardmäßig sind alle Menüeinträge nach ihrer Erzeugung aktiviert, daher muß die gewünschte Deaktivierung explizit erfolgen. ([Stelle im Code](#)).

Wird der Menüpunkt *disable say hello* angewählt, so zieht dies die Deaktivierung des Menüpunktes *say hello* und gleichzeitige (Re-)Aktivierung von *enable say hello* nach sich.

Die Selektierung des Menüpunktes *enable say hello* vollführt hingegen die duale Operation dazu, Aktivierung der Menüpunkte *say hello* und *disable say hello*. Aus diesem Grunde teilen sich beide Menüpunkte dieselbe Ereignisbehandlungs-Routine. Als Konsequenz dieser Forderung wird die Initialisierung der beiden `ActionListener` mit demselben Objekt notwendig ([Codestelle](#))-- daher kann die Umsetzung an dieser Stelle nicht mehr als anonyme innere Klasse realisiert werden, sondern erfolgt als „normale“ innere Klasse ([Codestelle](#)).

Die Anwahl des Menüpunktes *say hello* zeigt für eineinhalb Sekunden den Schriftzug *Hello!* am Bildschirm an. ([Ereignisbehandlungsroutine](#)).

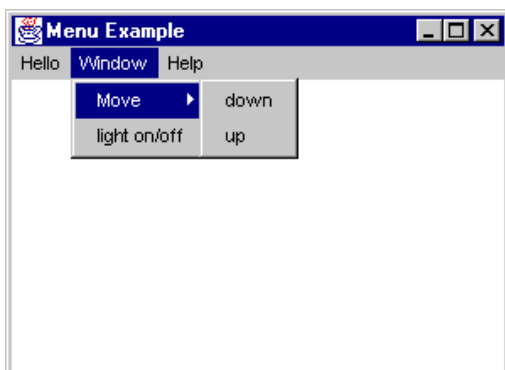


Abbildung 69: Das Menü Window (click on image to enlarge!)

Das zweite Menü der Menüleiste, mit *Window* betitelt, enthält ein Untermenü *Move* und eine Ausprägung von `CheckboxMenuItem`.

Die Ereignisbehandlung für Submenüeinträge erfolgt analog der auf höheren Ebenen; durch Definition des entsprechenden `ActionListener`-Instanz. ([Implementierung der Ereignisbehandlung des Menüpunktes up](#)). Bei Auswahl dieses Menüpunkts wird das Fenster, durch mehrmalige Änderung der Bildschirmkoordinaten in die linke obere Bildschirmecke verschoben.

Der Eintragsstyp `CheckboxMenuItem` erfordert eine Behandlungsroutine vom Typ `ItemListener`. Die Anzeige des Zustands wird durch die AWT automatisch vorgenommen, und bedarf keiner Anwenderinteraktion. ([Behandlungsroutine des Menüeintrages light on/off](#)). Der Menüpunkt ändert den Bildschirmhintergrund je nach Zustand von hell nach dunkel oder umgekehrt.

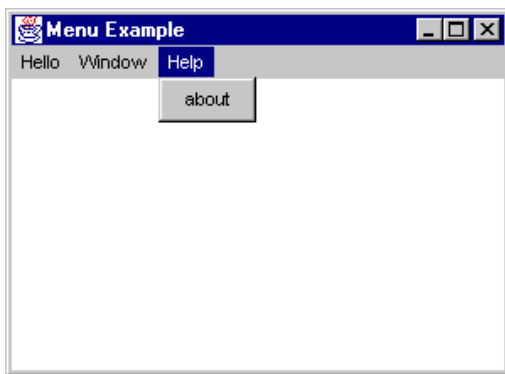


Abbildung 70: Das Menü Help (click on image to enlarge!)

Das dritte dargestellte Menü wurde als Standard-Hilfe-Menü deklariert. Existiert bereits ein Menü dieser Eigenschaft, so wird es durch das neu definierte ersetzt.

Die Aufnahme in die Menüleiste erfolgt nicht durch die bekannte add-Methode, sondern über den separaten Aufruf `setHelpMenu(Menu)` (Codestelle).

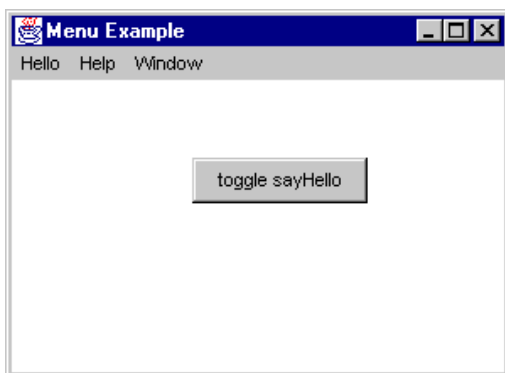


Abbildung 71: Das Kontextmenü (click on image to enlarge!)

Zusätzlich definiert die Applikation ein Kontextmenü welches die Funktionalität der Menüeinträge *enable say hello* und *disable say hello* vereinigt. Auch es nutzt die Ereignisbehandlungsroutine der beiden genannten Menüeinträge mit. (Codestelle).

Die Aktivierung und Anzeige am Punkt der Aktivierung muß durch den Anwender selbst realisiert werden. Hierfür ist ein entsprechender `MouseListener` umzusetzen. Die Methode `show(Component, int, int)` erlaubt hierfür die Übergabe von Koordinaten, an denen das Kontextmenü aufgeklappt wird.

Textfelder

Editierbare Textfelder für Benutzereingaben stellt die Klasse `TextField` bereit.

Zur Reaktion auf Tastenanschläge kann -- wie in [Übersicht](#) dargestellt -- jedes Objekt genutzt werden, das die von `Component` ererbte Schnittstelle `KeyListener` implementiert.

```
(1)import java.awt.Frame; (2)import java.awt.GridLayout; (3)import java.awt.Label; (4)import java.awt.
TextField; (5)import java.awt.Window; (6)import java.awt.event.KeyAdapter; (7)import java.awt.event.
KeyEvent; (8)import java.awt.event.WindowAdapter; (9)import java.awt.event.WindowEvent; (10) (11)public
class AWTE5 { (12) private static AWTE5 app; (13) private TextField tf_dm, tf_eur; (14) (15) public
static void main(String[] args) { (16) app = new AWTE5(); (17) } //main() (18) (19) public AWTE5() { (20)
Frame wnd = new Frame("Euro-Konverter"); (21) (22) wnd.setSize(200,100); (23) wnd.setLayout(new GridLayout
(2,2) ); (24) (25) Label lbl_dm = new Label ("DM"); (26) Label lbl_eur = new Label ("Euro"); (27) tf_dm =
new TextField("1", 7); (28) tf_eur = new TextField("1.95583", 7 ); (29) (30) wnd.add(lbl_dm); (31) wnd.add
(lbl_eur); (32) wnd.add(tf_dm); (33) wnd.add(tf_eur); (34) wnd.show(); (35) (36) (37) tf_dm.addKeyListener
( new KeyAdapter() { (38) public void keyReleased(KeyEvent e) { (39) tf_eur.setText( ""+Double.parseDouble
(tf_dm.getText())*1.95583 ); (40) } //keyPressed() (41) } //anonymous inner class (42) ); (43) (44) tf_eur.
addKeyListener( new KeyAdapter() { (45) public void keyReleased(KeyEvent e) { (46) tf_dm.setText(""+Double.
parseDouble(tf_eur.getText())/1.95583 ); (47) } //keyPressed() (48) } //anonymous inner class (49) ); (50)
(51) wnd.addWindowListener( new WindowAdapter() { (52) public void windowClosing(WindowEvent we) { (53)
Window wnd = we.getWindow(); (54) (55) wnd.setVisible(false); (56) wnd.dispose(); (57) System.exit(0);
(58) } //windowClosing() (59) } //anonymous inner class (60) ); (61) } //constructor (62)} //end class AWTE5
```

Beispiel 96: Ein einfacher Euro-Umrechner [AWTE5.java](#)

Bildschirmausgabe:



Abbildung 72: Ein Zwischenberechnungsstand (click on image to enlarge!)

Die Applikation nutzt das [GridLayout](#) zur Ausrichtung der vier visuellen Komponenten. Hierfür wird der entsprechende Layoutmanager dahingehend parametrisiert, daß er ein quadratisches Layout mit jeweils zwei horizontalen und vertikalen Einträgen erlaubt. ([Codestelle](#)).

Die Beschriftungen der Spalten sind als Labels realisiert.

Zum Erzeugungszeitpunkt wird den Text-Feldern ein Startwert, und die horizontale Ausdehnung von sieben Zeichen zugewiesen. ([Codestelle](#)).

In den (anonymen inneren) Klassen zur Ereignisbehandlung, wird die Methode [keyTyped](#) ([KeyEvent](#)) überschrieben. Sie wird *nach* Abschluß des Tastendrucks aufgerufen.

In der Implementierung dieser Methode wird die tatsächliche Umrechnung vorgenommen, und daß Umrechnungsergebnis im jeweils anderen Feld ausgegeben. ([Codestelle](#)).

Hinweis: Eine Behandlung des möglicherweise generierten `NumberFormatException`-Ausnahmeereignisses findet aus Übersichtlichkeitsgründen nicht statt.

Dialoge

Ausprägungen der Klasse [Dialog](#) stellen eine Möglichkeit zur Realisierung einfacher Ein- und Ausgabefenster dar. Diese Art Fenster eignet sich besonders für kurze Anfragen, oder Meldungen, an den Anwender.

[Beispiel `awtEx4`](#) enthält eine solche Nachrichtenbox. Dialogfenster können, im Gegensatz zu den bekannten Applikationsfenstern, *modal* sein. Dies bedeutet, daß ein solches Fenster nicht durch den Anwender in den Hintergrund versetzt werden kann; es erzwingt eine Reaktion -- zumeist in Form der Bestätigung einer Meldung o.ä. -- bevor die Applikation fortfährt.

▲ 3.2.8 Swing

Mit dem JDK v1.1 wurde als zusätzliche API zur Erstellung von graphischen Oberflächen [Swing](#) vorgestellt. Bis zur aktuellen Version 1.3 ist diese API weiterhin Bestandteil des [Extension Paketes](#), und steht daher nicht auf allen Plattformen zur Verfügung. Des weiteren können sich noch zukünftige Änderungen an den derzeit publizierten APIs ergeben, so daß bestehender Code überarbeitet werden muß.

Folgende Überlegungen führten zur Entwicklung von Swing:

- Durch die Übernahme des plattformspezifischen look-and-feels der AWT-Applikationen entsteht teilweise großer Portierungsaufwand zur Realisierung nicht-nativ verfügbarer Primitive.
- Jede Änderung an den realen GUIs der Zielsysteme läßt Modifikationen an der AWT-Implementierung notwendig werden.
Dies konterkariert die angestrebte Plattformunabhängigkeit auf der Ebene der angebotenen GUIs zunehmend.
- Aufwendige und komplexe Benutzeroberflächen sind mit dem AWT u. U. sehr schwer zu realisieren.

Daher wurde mit den *Java Foundation Classes*, deren Bestandteil Swing ist, versucht eine echte Alternative zum bestehenden -- und weiter angebotenen -- AWT zu schaffen. Dabei erhöht Swing massiv die Anzahl der angebotenen visuellen Komponenten, und fügt einige sehr mächtige Primitive wie beispielsweise zur Darstellung von Bäumen hinzu.

Anders als die AWT setzt Swing nicht auf den Möglichkeiten des zugrundeliegenden GUI-Systems auf, sondern stellt selbst die gesamte Verwaltung und Verarbeitung der angebotenen Primitive zur Verfügung. Als Folge dieses Ansatzes realisiert Swing ein eigenes look-and-feel, das auf allen unterstützten Plattformen unverändert präsentiert wird. Mit diesem -- *Metal* genannten -- Layout entsteht erstmals ein typisches Aussehen nativer Java-Applikationen.

Technisch ist Swing mit den seit JDK v1.1 angebotenen *leichtgewichtigen Komponenten*, den Java Beans, realisiert. Die Realisierung graphischer Primitivoperationen erfolgt daher nicht mehr durch Operationen des zugrundeliegenden GUI-Systems, sondern durch die angebotenen Swing-Komponenten selbst.

Dies führt dazu, daß die gesamte Swing-API ohne native Methoden -- vollständig in Java codiert --

realisiert werden konnte. Hieraus ergeben sich weitere Vorteile in der Anwendung, insbesondere in der Fehlersuche.

Jedoch führt die verfolgte *pure Java*-Implementierung auch zu Problemen. So erhöht sich durch den de-facto emulierenden Ansatz der Speicherplatzbedarf, da nicht mehr auf die evtl. durch die Plattform angebotenen Routinen zurückgegriffen wird. Flankierend sinkt die Ausführungsgeschwindigkeit durch die gestiegene Menge an auszuführendem Java-Code.

Anmerkung:

Aus rein praktischen Erwägungen sprechen derzeit noch zwei weitere Punkte gegen den breiten Swing-Einsatz: zunächst der Reifegrad und die Entwicklungsstabilität. Als Bestandteil des Java Extension Paketes ist Swing explizit als experimenteller Bestandteil der derzeit verfügbaren Java-API gekennzeichnet. In der verbreiteten Implementierung finden sich noch kleinere und größere Fehler und Ungereimtheiten, die gegen die Verwendung in Produktivapplikationen sprechen. Wie bei allen anderen Bestandteilen des *javax*-Paketes auch, behält sich SUN explizit das Recht vor Schnittstellen und Funktionalität ohne Ankündigung zu verändern oder aus dem Angebot herauszunehmen. Daher verbietet sich, unter Berücksichtigung einer zukünftigen Pflege des entstehenden Applikationscodes, die Nutzung dieser Klassen schon fast.

Ergänzend sei noch erwähnt, daß seitens der aktuell verfügbaren Web-Browser noch keine Swing-Unterstützung umgesetzt ist. Dieses Manko läßt sich zwar durch die zusätzliche manuelle Installation des [Java-Plugins](#) beheben, verlangt dem Anwender jedoch einen zusätzlichen Installationsvorgang ab.

Swing verwirklicht durchgängig das von T. Renskaug initiierte *Model-View-Controller*-Konzept. Dessen hervorstechendstes Kennzeichen die Separierung des Codes in drei verschiedene Verwendungskategorien ist:

- **Model** -- Enthält die Daten und den Zustand des Dialogelements
- **View** -- Übernimmt die graphische Darstellung der visuellen Komponente
- **Controller** -- Verbindet *Model* und *View*; empfängt Ereignisse und leitet entsprechende Maßnahmen an *Model* und *View* ein

Die gesamte Verarbeitungslogik ist dabei in der *Model*-Klasse zentralisiert. Zu jedem *Model* können dabei gleichzeitig verschiedene *View*-Klassen existieren.

Swing reduziert diesen Mechanismus auf zwei verschiedenen Klassen-Typen. Hierbei wird die *View*- und die *Controller*-Komponente zu einer Einheit verschmolzen. Das entstehende Design wird *Model-Delegate-Prinzip* genannt.

```
(1)import java.awt.Window;
(2)import java.awt.event.ActionEvent;
(3)import java.awt.event.ActionListener;
(4)import java.awt.event.WindowAdapter;
(5)import java.awt.event.WindowEvent;
(6)import javax.swing.JButton;
(7)import javax.swing.JFrame;
(8)import javax.swing.JPanel;
(9)import javax.swing.SwingUtilities;
(10)import javax.swing.UIManager;
(11)import javax.swing.UnsupportedLookAndFeelException;
(12)
(13)public class SwingEx1 extends JFrame implements ActionListener {
(14)    public SwingEx1() {
(15)        super("first swing application");
(16)        JPanel myPanel = new JPanel();
(17)        JButton btn_win = new JButton("Windows");
(18)        JButton btn_motif = new JButton("Motif");
(19)        JButton btn_metal = new JButton("Metal");
(20)
(21)        btn_win.setToolTipText("switch to Windows look-and-feel");
(22)        btn_motif.setToolTipText("switch to Motif look-and-feel");
(23)        btn_metal.setToolTipText("switch to Metal look-and-feel");
(24)
(25)        myPanel.add( btn_win );
(26)        myPanel.add( btn_motif );
(27)        myPanel.add( btn_metal );
(28)
(29)        btn_win.addActionListener(this);
(30)        btn_motif.addActionListener(this);
(31)        btn_metal.addActionListener(this);
(32)
(33)        getContentPane().add("South", myPanel);
```

```

(34)
(35)         addWindowListener( new WindowAdapter() {
(36)             public void windowClosing(WindowEvent we) {
(37)                 Window wnd = we.getWindow();
(38)
(39)                 wnd.setVisible(false);
(40)                 wnd.dispose();
(41)                 System.exit(0);
(42)             } //windowClosing()
(43)         } //anonymous inner class
(44)     );
(45) } //constructor
(46)
(47) public void actionPerformed(ActionEvent ae) {
(48)     String cmd = ae.getActionCommand(),
(49)           plaf="";
(50)
(51)     if ( cmd.equals("Metal") ) {
(52)         plaf = "javax.swing.plaf.metal.MetalLookAndFeel";
(53)     } else {
(54)         if ( cmd.equals("Motif") ) {
(55)             plaf = "com.sun.java.swing.plaf.motif.
MotifLookAndFeel";
(56)         } else {
(57)             if( cmd.equals("Windows") ) {
(58)                 plaf = "com.sun.java.swing.plaf.windows.
WindowsLookAndFeel";
(59)             } //else
(60)         } //else
(61)     } //else
(62)
(63)     try {
(64)         UIManager.setLookAndFeel(plaf);
(65)         SwingUtilities.updateComponentTreeUI(this);
(66)     } catch (UnsupportedLookAndFeelException e) {
(67)         System.err.println(e.toString());
(68)     } catch (ClassNotFoundException e) {
(69)         System.err.println(e.toString());
(70)     } catch (InstantiationException e) {
(71)         System.err.println(e.toString());
(72)     } catch (IllegalAccessException e) {
(73)         System.err.println(e.toString());
(74)     } //catch
(75) } //actionPerformed()
(76)
(77) public static void main(String[] args) {
(78)     SwingEx1 wnd = new SwingEx1();
(79)     wnd.setSize(300,200);
(80)     wnd.pack();
(81)     wnd.setVisible(true);
(82) } //end main()
(83)} //class SwingEx1

```

Beispiel 97: Eine erstes Swing-Applikation [SwingEx1.java](#)

BildschirmAusgabe:



Abbildung 73: Das Beispielprogramm im MS-Windows-Layout *(click on image to enlarge!)*

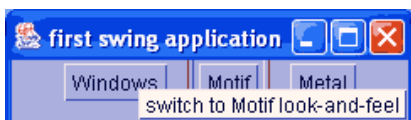


Abbildung 74: ... im Motif-Layout *(click on image to enlarge!)*



Abbildung 75: ... im Metal-Layout (click on image to enlarge!)

Diese Swing-Applikation definiert ein einfaches Fenster mit drei Buttons.

Deutlichstes Kennzeichen der Nutzung von Swing ist das Präfix `J` vor den bekannten AWT-Komponentennamen. So erweitert die Applikation das Swing-Analogon der AWT-Klasse `Frame`, jetzt unter dem Namen `JFrame`.

Die Schaltflächen sind entsprechend Ausprägungen von `JButton`. Mit der Methode `setToolTipText(String)` wird die rein Swing-interne Möglichkeit zur Definition eines Hilfstexts genutzt, der bei Überstreichen einer Komponente mit der Maus automatisch angezeigt wird. Die Implementierung greift nicht auf Mechanismen des zugrundeliegenden GUI-Systems zurück, sondern ist vollständig innerhalb von Swing realisiert.

Die Ereignisbearbeitung ist identisch zur AWT v1.1 realisiert, und kann daher nahezu unverändert übernommen werden.

Als Behandlungsroutine der drei Schaltflächen ist der Wechsel des Bildschirmlayouts zur Laufzeit realisiert. Nach Aktivierung der entsprechenden Schaltfläche wird das entsprechende, innerhalb der Swing-API vordefinierte Layouts referenziert, und durch die statische Methode `setLookAndFeel(String)` geladen. Nach der Aktualisierung der Bildschirmdarstellung präsentiert sich die gesamte Applikation im neuen Layout.

Service provided by [Mario Jeckle](#)

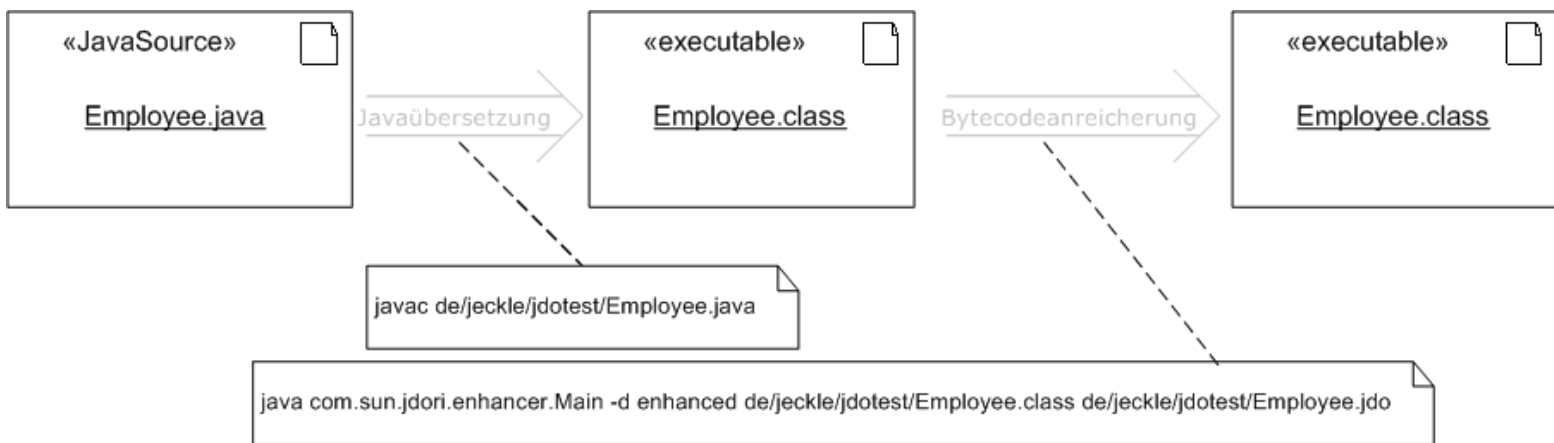
Generated: 2004-01-08T22:57:43+01:00

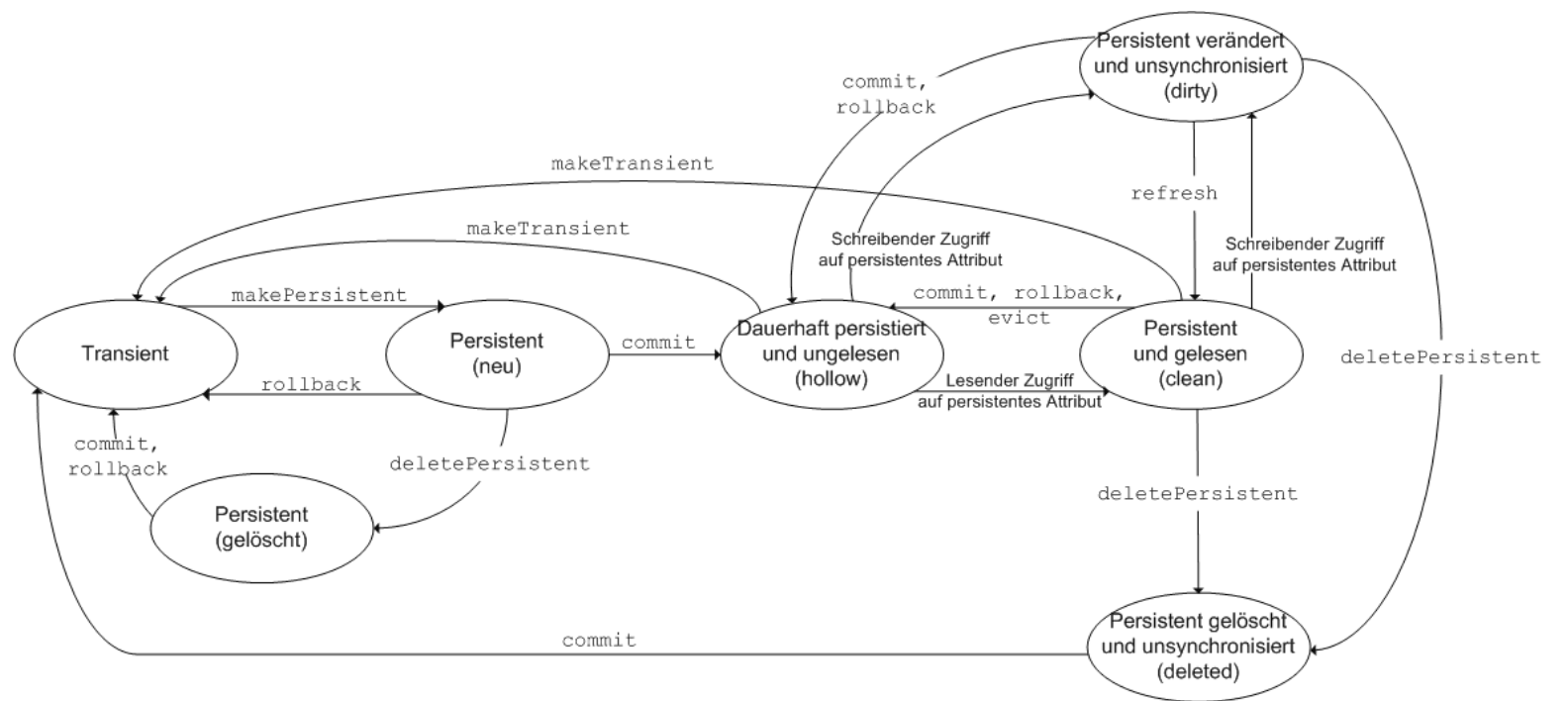
► [Feedback](#)

► [SiteMap](#)

► This page's original location: <http://www.jeckle.de/vorlesung/java/script.html>

► [RDF description for this page](#)





```
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;
import java.util.Vector;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;

import de.jeckle.jdotest.Employee;

public class JDOStoreObj {
    public static void main(String args[]) {
        Properties props = new Properties();
        try {
            InputStream is =
                ClassLoader.getResourceAsStream("jdo.properties");
            props.load(is);
        } catch (IOException ioe) {
            System.out.println("Error loading properties");
            System.exit(1);
        }
        PersistenceManagerFactory pmf =
            JDOHelper.getPersistenceManagerFactory(props);
        PersistenceManager pm = pmf.getPersistenceManager();

        Vector empCol = new Vector();

        Employee emp1 = new Employee();
        emp1.setName("Mario Jeckle");
        emp1.setDepartment("D001");
        emp1.addProject("P001");
        emp1.addProject("P002");
        empCol.add(emp1);

        Employee emp2 = new Employee();
        emp2.setName("John DoeX");
        emp2.setDepartment("D003");
        emp2.addProject("P001");
        emp2.addProject("P042");
        empCol.add(emp2);

        Employee emp3 = new Employee();
        emp3.setName("John Doe");
        emp3.setDepartment("B042");
```

```
empCol.add(emp3);
```

```
Employee emp4 = new Employee();  
emp4.setName("Barnie Bar");  
emp4.setDepartment("B042");
```

```
pm.currentTransaction().begin();  
pm.makePersistentAll(empCol);  
pm.makePersistent(emp4);
```

```
pm.currentTransaction().commit();  
pm.close();
```

```
}
```

```
}
```

```
import java.io.IOException;
import java.io.InputStream;
import java.util.Iterator;
import java.util.Properties;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;

import de.jeckle.jdotest.Employee;

public class JDORollback {

    public static void main(String args[]) {
        Properties props = new Properties();
        try {
            InputStream is =
                ClassLoader.getResourceAsStream("jdo.properties");
            props.load(is);
        } catch (IOException ioe) {
            System.out.println("Error loading properties");
            System.exit(1);
        }
        PersistenceManagerFactory pmf =
            JDOHelper.getPersistenceManagerFactory(props);
        PersistenceManager pm = pmf.getPersistenceManager();

        Employee e = new Employee();
        e.setName("Marta Mayer");

        pm.currentTransaction().begin();
        pm.makePersistent(e);
        pm.currentTransaction().commit();
        displayPersistedObjects(pm);

        System.out.println("Martha gets married and changes her name");

        pm.currentTransaction().begin();
        e.setName("Marta Smith");
        pm.makePersistent(e);
        displayPersistedObjects(pm);
        System.out.println("Suppose an error happens now ... \nRolling back");
        pm.currentTransaction().rollback();

        displayPersistedObjects(pm);
    }
}
```

```
    }  
    private static void displayPersistedObjects(PersistenceManager pm) {  
        Iterator i = pm.getExtent(Employee.class, false).iterator();  
        while (i.hasNext()) {  
            System.out.println((Employee) i.next());  
        }  
    }  
}
```

```
import java.io.IOException;
import java.io.InputStream;
import java.util.Iterator;
import java.util.Properties;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;

import de.jeckle.jdotest.Employee;
public class JDONonTransact {
    public static void main(String args[]) {
        Properties props = new Properties();
        try {
            InputStream is =
                ClassLoader.getResourceAsStream("jdo.properties");
            props.load(is);
        } catch (IOException ioe) {
            System.out.println("Error loading properties");
            System.exit(1);
        }
        PersistenceManagerFactory pmf =
            JDOHelper.getPersistenceManagerFactory(props);
        PersistenceManager pm = pmf.getPersistenceManager();

        Employee e = new Employee();
        e.setName("Marta Mayer");
        pm.currentTransaction().setNontransactionalWrite(true);

        pm.currentTransaction().begin();
        pm.makePersistent(e);
        pm.currentTransaction().commit();
        displayPersistedObjects(pm);

        //martha gets married and changes her name

        e.setName("Marta Smith");

        displayPersistedObjects(pm);
    }
    private static void displayPersistedObjects(PersistenceManager pm) {
        Iterator i = pm.getExtent(Employee.class, false).iterator();
        while (i.hasNext()) {
            System.out.println((Employee) i.next());
        }
    }
}
```

```
}  
}
```

```
import java.io.IOException;
import java.io.InputStream;
import java.util.Iterator;
import java.util.Properties;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;

import de.jeckle.jdotest.Employee;

public class JDOListObj {
    public static void main(String args[]) {
        Properties props = new Properties();
        try {
            InputStream is =
                ClassLoader.getResourceAsStream("jdo.properties");
            props.load(is);
        } catch (IOException ioe) {
            System.out.println("Error loading properties");
            System.exit(1);
        }
        PersistenceManagerFactory pmf =
            JDOHelper.getPersistenceManagerFactory(props);
        PersistenceManager pm = pmf.getPersistenceManager();

        Iterator i = pm.getExtent(Employee.class, false).iterator();
        while (i.hasNext()) {
            System.out.println((Employee)i.next());
        }
    }
}
```



```
import java.io.IOException;
import java.io.InputStream;
import java.util.Collection;
import java.util.Iterator;
import java.util.Properties;

import javax.jdo.Extent;
import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.Query;

import de.jeckle.jdotest.Employee;
public class JDOQuery {
    public static void main(String args[]) {
        Properties props = new Properties();
        try {
            InputStream is =
                ClassLoader.getResourceAsStream("jdo.properties");
            props.load(is);
        } catch (IOException ioe) {
            System.out.println("Error loading properties");
            System.exit(1);
        }
        PersistenceManagerFactory pmf =
            JDOHelper.getPersistenceManagerFactory(props);
        PersistenceManager pm = pmf.getPersistenceManager();

        Extent ext = pm.getExtent(Employee.class, false);
        String filter = "department == \"B042\"";
        Query qry = pm.newQuery(ext, filter);
        qry.setOrdering("name ascending");
        qry.compile();
        Collection c = (Collection) qry.execute();

        Iterator i = c.iterator();
        while (i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

```
import java.io.IOException;
import java.io.InputStream;
import java.util.Collection;
import java.util.Properties;

import javax.jdo.Extent;
import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.Query;

import de.jeckle.jdotest.Employee;

public class JDODeleteObj {
    public static void main(String args[]) {
        Properties props = new Properties();
        try {
            InputStream is =
                ClassLoader.getResourceAsStream("jdo.properties");
            props.load(is);
        } catch (IOException ioe) {
            System.out.println("Error loading properties");
            System.exit(1);
        }
        PersistenceManagerFactory pmf =
            JDOHelper.getPersistenceManagerFactory(props);
        PersistenceManager pm = pmf.getPersistenceManager();

        Extent ext = pm.getExtent(Employee.class, false);
        String filter = "name == \"Marta Smith\"";
        Query qry = pm.newQuery(ext, filter);
        Collection c = (Collection) qry.execute();

        pm.currentTransaction().begin();
        pm.deletePersistentAll(c);
        pm.currentTransaction().commit();
        System.out.println("Object deleted");
    }
}
```

```
javax.jdo.PersistenceManagerFactoryClass=com.triactive.jdo.PersistenceManagerFactoryImpl
javax.jdo.option.ConnectionURL=jdbc:mysql://localhost/jdotest/
javax.jdo.option.ConnectionDriverName=com.mysql.jdbc.Driver
javax.jdo.option.ConnectionUserName=mario
javax.jdo.option.ConnectionPassword=thePassword
com.triactive.jdo.autoCreateTables=true
```

Java Database Connectivity

Enterprise Java Bean
(bean managed persistence)

Java Data Objects

Transparenz
im Zugriff

Eingriffsmöglichen
für den Programmierer

Welcome to the TJDO Project Page

Links

[SourceForge Site](#)[Download](#)[Tutorial](#)[Docs](#)[JDO Javadocs](#)[TJDO Javadocs](#)[TJDO Roadmap](#)[Contributing Guidelines](#)[Browse CVS](#)[Sun's JDO Site](#)

Overview

TriActive JDO (TJDO) is an open source implementation of Sun's JDO specification ([JSR 12](#)), designed to support transparent persistence using any JDBC-compliant database. Although the project is formally in beta release, TJDO is currently deployed and running successfully in a number of commercial installations.

The current version has been [tested successfully](#) using Cloudscape, DB2, Firebird, MySQL, Oracle 8i, PostgreSQL, SAP DB, and MS SQL Server.

Features

- Supports JDO 1.0.1.
- Implements the entire JDOQL query language, including several [useful method enhancements](#).
- Auto-creates all necessary schema elements (tables, foreign keys, indexes) according to your app's classes and JDO metadata.
- Auto-validates expected schema structure at runtime, reducing failures from schema evolution.
- Can map Java classes to [SQL views](#), and allows for [direct SQL queries](#), to leverage SQL capabilities not normally accessible through JDO.
- Designed to be lightweight and fast.

Design Goals

As an O-R mapper TJDO falls under the category of *schema generators*, meaning it takes your user-written Java classes and, to the degree possible, fully automates the tasks required to transparently persist objects to a database. Although there are

several ways to "tweak" the mapping strategy, for the most part the developer delegates these tasks to the JDO layer. Schema generators are best suited to projects whose persistent data model is being newly developed, or perhaps fully refactored, and whose developers would prefer to design mostly from an object rather than a relational point of view.

If things like DBMS portability and rapid development turnaround are more important to you than fine-grained control over exact data representations, then TJDO is probably what you're looking for. Conversely, if you're one of those who tend to spend long hours fussing over where you should "denormalize for performance", it probably is not.

Getting Started

New users can start by [downloading the latest version](#) and including the required jar files in their classpath. The [tutorial](#), [documentation](#), and the [JDO API Javadocs](#) are good places to begin when exploring TJDO. If you have any questions, comments, or problems please consult the documentation or use the [TJDO forums on SourceForge](#).

Being an open source project, we are always interested in finding people to help with the project. If you are interested in contributing, contact the [TJDO Project Administrator](#). You can find information about tasks, bugs, obtaining source code, etc... on the [TJDO SourceForge Site](#).

We hope you find TJDO useful. Please [let us know](#) what you think of it.

News

November 11, 2003

TJDO 2.0 Final was released today! [Binary](#) and

[source](#) distributions are available for this release. This is the first TJDO release to be labeled as stable. Many new features have been added to the release, and many bugs have been fixed. Notable changes are: JDO 1.0.1 support, package-level metadata files, support for fields of type HashSet, the ability to pass persistent and transient Sets as parameters into Querys, the option to turn off table and constraint validation, the ability to utilize cascaded deletes, allowing non-nullable owner fields for inverse collections, RestoreValues mode, use of a validating XML parser, work towards passing the TCK test suite, many bug fixes, and much more. See the [release notes](#) for a full list of the changes.

Also, check out the [TJDO roadmap](#) to get an idea of where we are heading. If you are interested in contributing to TJDO, please read the [contributing guidelines](#). As always, you can post to our [forums](#) and let us know what you think.

March 18, 2003

TJDO 2.0 Beta3 has been released! [Binary](#) and [source](#) distributions are available for this development build. Most prevalent in this release is support for **many** more databases, including: PostgreSQL 7.3, MySQL, MS SQL Server 7.0 and 2000, DB2 7.2.3, SAP DB 7.4.3, and full support for DB2J (Cloudscape 5). Additionally, several bugs have been fixed. See the [release notes](#) for more details.

Keep an eye out for TJDO 2.0 Final ... we hope to get this stable major release out soon (after the bugs get shaken out of TJDO 2.0 Beta3). Also on the horizon are: JCA support, additional default persistent types (including Lists and more of the collections classes), working towards passing the JDO TCK, a redesigned TJDO home page, and a project name change!

For now, try running TJDO 2.0 Beta3 on your

favorite [supported database](#), and [let us know](#) what you think.

January 22, 2003

TJDO 2.0 Beta2 has been released! [Binary](#) and [source](#) distributions are available for this development build. This release fixes some bugs and adds new functionality. Most notably added was support for PostgreSQL, Firebird, and Cloudscape 5.0 for data persistence. See the [release notes](#) for more details.

The next planned release is TJDO 2.0 Beta3, which will contain J2EE connector support, support for MySQL, and other minor enhancements. TJDO 2.0 Beta3 will serve as a release candidate and will be followed shortly by TJDO 2.0. Give this release a try, and please [let us know](#) of any problems.

September 26, 2002

We have added a TJDO tutorial to the web page. This is geared to help beginners write their first TJDO program. This is a good starting place for anyone looking to learn how TJDO works. Keep your eye out for more tutorials with more complex examples, but for now you can chew on [this](#).

September 9, 2002

TJDO 2.0 Beta was released today! This version is compliant with the JDO 1.0 spec, and contains other enhancements. Download it [here](#), and don't forget to read the [news release](#).

Once the bugs are worked out of the beta, we will release the first stable version - 2.0. Stay tuned for more information...

August 14, 2002

The first goal of this project is to [bring TJDO up to the 1.0 spec](#). This will allow us to use Sun's

reference enhancer and make TJDO a viable option for a JDO implementation. The changes between the 0.95 spec and the 1.0 spec are outlined [here](#). A [document](#) has been put together outlining the steps involved in resolving these differences. Work will soon commence to bring TJDO up to the 1.0 spec.

Direct questions/comments to the [TJDO Project Administrator](#).

Last Updated: \$Date: 2003/11/11 20:30:03 \$





The Source for Java Developers



Developers Home > Products & Technologies > Java Technology > J2SE > Core Java
>

Profile and Registration | Why Register?

Core Java

Java Data Objects (JDO)

Downloads

Reference

- Documentation
- FAQs

Community

- Bug Database
- Forums

Learning

- Tutorial and Code Camps

The Java Data Objects (JDO) API is a standard interface-based Java model abstraction of persistence, developed as Java Specification Request 12 [JSR 12](#) under the auspices of the [Java Community Process](#). If you are an application programmer, you can use JDO technology to directly store your Java domain model instances into the persistent store (database). » [Read More](#)

Benefits of Using JDO for Application Programming

Applications written with the JDO API are portable, and independent of the underlying database. You can focus on your domain object model and leave the details of persistence (field-by-field storage of objects) to the JDO implementation.

What's New

September 2003

JDO Specification (JSR 12) Download the final release of the spec from the Java Community Process (JCP) site. JDO 1.0.1 has been released.

Community

JDO Central

Online resource for JDO developers. Find out the latest on JDO from experts, fellow users, and vendors. Chats, news, and more.

Events

2004 JavaOne Conference. San Francisco , CA Be there!

Join the thousands of developers worldwide who come to the JavaOne conference each year in San Francisco to immerse themselves in Java technology, the latest innovations, the community, and the learning opportunities. » [Read More](#)

Subscribe to Newsletters. Members of [Sun Developer Network](#) can [sign up](#) to receive these (and other) newsletters. Not yet a member? [Join us!](#)

Java Technology Fundamentals

New to Java? Learn the basics of the Java programming language and keep up-to-date on additions to the New-to-Java Programming Center.

Core Java Technologies Newsletter

Find out about new enterprise Java technologies, products, tools, and resources for developers.

Core Java Technologies Tech Tips

Get expert tips, sample code solutions, and techniques for developing in the Java 2 Platform, Standard Edition (J2SE).

» [Read More](#)

Related Links

Popular Downloads

- J2SE 1.4.2
- Java Web Services Developer Pack 1.3

Technical Topics

- Performance
- Web Services
- Security
- Desktop

Products and Technologies

- J2SE 1.5 (coming soon)
- J2SE 1.4.2
- Desktop Java
- Java Web Services
- J2EE
- Java Dynamic Management Kit (JDMK)
- Sun Java Studio Standard IDE

Sun Resources

- [java.net](#)
- [Java Upgrade Program](#)
- [New to Java Center](#)
- [Professional Certification](#)
- [Professional Training](#)
- [JavaOne Online](#)



Raghavan "Rags" Srinivas,
Staff Engineer, Developer and
Security Expert



[Company Info](#) | [About SDN](#) | [Press](#) | [Contact Us](#) |
[Employment](#)
[How to Buy](#) | [Licensing](#) | [Terms of Use](#) | [Privacy](#) |
[Trademarks](#)

Copyright 1994-2004 Sun Microsystems, Inc.

**A Sun Developer Network
Site**

Unless otherwise licensed,
code in all technical manuals
herein (including articles,
FAQs, samples) is provided
under this [License](#).

 [Content Feeds](#)



Login

Password

[Login](#) [Join Now](#)

[Forgot your password?](#)

Search JDOcentral.com

- [JDO Success Stories](#)
- [JDO Books](#)
- [Meet the JDO Experts](#)
- [JDO Vendor Directory](#)
- [About Us](#)
- [Contact Us](#)

Charter Sponsors



Latest Commentary [\[More \]](#)

JBoss Chooses Java Data Objects

Several months ago, Marc Fleury announced that JBoss would make their CMP implementation available to developers working on persistence solutions outside the server environment. With no details, rumors flew about what this actually meant for persistence and the open source community. [\[More \]](#)

JDO or CMP?

Java Data Objects (JDO) and Enterprise JavaBeans (EJB) Container Managed Persistence (CMP) were developed concurrently, but took different paths. To see where each technology is applicable, read the following [article](#). You can also read a Chinese translation of this article by clicking [here](#). Check out the discussion of this article on [TheServerSide](#) and support JDO by adding your own views to the [discussion thread](#).

2004 Featured Events [\[More \]](#)

1/06	Introduction to the JDO Standard - Madison, WI
1/07	Introduction to the JDO Standard - Milwaukee, WI
1/08	Introduction to the JDO Standard - West Schaumburg, IL
1/13	Introduction to the JDO - WebEx Presentation
1/13	Introduction to the JDO Standard - Dallas, TX
1/13	Java Data Objects (JDO) - Dublin, OH
1/26 - 1/28	JDO Hands-On Workshop - Huntington Beach, CA
1/28	Introduction to the JDO Standard - Indianapolis, IN
1/29	Object Persistence and the JDO API - London, UK

In the News [\[More \]](#)

12/15	Versant Corp. Becomes Charter Member of JDOcentral.com NEW
12/12	Riflexo Releases JCreDO JDO Professional & Enterprise Editions NEW
12/11	SolarMetric Releases Latest Version of Popular Kodo JDO Product: Introduces Preview Features of the JDO 2.0 Standard NEW
10/29	JInspired Announces JDBInsight 2.0 Release
10/08	Versant Delivers JDO for Versant Object Database
10/07	Riflexo Released JCreDO 1.0, Free Compliant JDO Implementation
9/30	Versant Partners with SolarMetric to Deliver Standards-Based JDO Solutions
9/28	Versant Signs Definitive Agreement to Merge with Poet Holdings, Inc.
9/9	Riflexo Releases JCreDO Beta with JBuilder Integration and Full Web Application Code Sample
8/12	Sun Microsystems Supports Java Data Objects (JDO) And Becomes Charter Member of JDOCentral.com

Whitepapers [\[More \]](#)

- [Best Practices for JDO Performance](#) - FastObjects **NEW**
- [JDOQL: The JDO Query Language](#) - David Jordan/Object Identity, Inc.
- [A Comparison Between Java Data Objects \(JDO\), Serialization and JDBC for Java Persistence](#) - David Jordan/Object Identity, Inc.
- [Buy Versus Build](#) - Neelan Choski/SolarMetric, Inc.
- [Java Data Objects - The Future for Java Object Persistence](#) - Keiron McCammon/Versant Corporation

- [Why JDO is a Critical Component of J2EE?](#) - LiBeLiS
- [OpenFusion - Java Data Objects](#) - PrismTech

Articles [\[More \]](#)

New Content Around the Site



JDO Meets Maven

Maven is a tool for managing and building projects providing an alternative to the accepted Ant build process. Maven is almost totally plug-in-driven, and provides plug-ins for many common tasks (for example: building EJB components such as WARs and EARs, generating documentation, running unit tests) and related software (for example: Checkstyle, PMD, Clover, JCoverage). [\[More \]](#)



JDO 2.0: Lot's of Changes

Discussed at the Kickoff Meeting

"...Working groups with competing vendors got together over the 3 day meeting to work on things. Vendors were discussing what they did right now, and what they were planning to do in the future. I probably learnt much about the different JDO vendor products just via this information alone..." [\[More \]](#)



Dedicated JDO Books Page Now Available

More and more JDO books are hitting the book shelves. You can find an overview what is available today right here... [\[More \]](#)



JDO Success Story: Building Multimedia Informatics Systems Using JDO

Multimedia Informatics: Learn How JDO and FastObjects are being used at UCLA. [\[More \]](#)



JDO Success Story: CastorTPV 1.2 - A Case Study of a JDO Based Application

Advanced Quality Solutions is a Spanish IT provider whose portfolio includes commercial software products, custom development, and system maintenance. [\[More \]](#)

More JDO Success Stories... [\[More \]](#)

JDO Expert Corner

Sample Of a Persistence Capable Class Wrapping an Image In a Gallery

(by Stephen Kolaroff)

This example demonstrates a strategy for dealing with user-specified pictures (or "avatars") and photos, and how to store them in a gallery. The class itself may be re-used in different contexts with quite little effort. We will generalize this strategy and provide hints about its applicability. The code example, which is available on JDO Central Code Exchange page, is an extract of a full JDO code sample showing a web application called "Senat" discussion forum (<http://jcredo.com/senat>), which full code can be downloaded together with JCreDO - Riflexo's free JDO implementation - on jcredo.com.



Creating and Using Simple Extended Optimistic Transactions

(by David Ezzio)

JDO 1.0 provides two kinds of transactions, but both assume that the persistence manager is open during the duration of the transaction. In web and enterprise applications, the persistence manager may close between the view and the update requests. In these architectures, the application may need a technique to verify that the version of the object when changed is the same as the version when viewed.



Persistent Identity: The Persistent Object's Essential Attribute (by David Ezzio)

Persistent identity is the essential attribute of persistent objects. JDO provides both datastore identity and application identity.

Because JDO makes datastore identity opaque, the application needs a technique to capture it. The data

FrontierSuite for JDO

Develop
Save upto 70% time

Run
Imp. performance upto 400%

JDO 1.0 Compliant

Distributed Caching

Class Enhancement

Reverse Engineering

Forward Engineering

Bridge Pattern

Code Generation

O/R Mapping

Learn More

ObjectFrontier
Tangible Business Benefits through Open Standard Technologies

- 11/21 [JDO 2.0 Preview: Java Object Persistence Made Easy](#) 
- 11/17 [JDO Meets Maven](#)
- 9/30 [Versant Includes JDO in Object Database](#)
- 9/15 [Sun Back on JDO Bandwagon](#)
- 9/9 [Sample Of a Persistence Capable Class Wrapping an Image In a Gallery](#)
- 8/21 [JDO 2.0: Lot's of Changes Discussed at the Kickoff Meeting](#)
- 7/25 [JDBC, CMP, or JDO?](#)
- 7/21 [Factor in Performance and Portability](#)
- 7/18 [More Object Persistence Choice Factors](#)
- 7/16 [Choose the Right Object Persistence](#)
- 7/10 [JDO Brings DB Programming into 21st Century Despite Controversy](#)

object's equals method uses the persistent identity to determine when two memory objects refer to the same persistent object.

Hot Forum Topics



Use Cases for JDO 2.0 - The Brainstorming Thread

"Just talking about "object-persistence" in the abstract is all well and good. But, running JDO through the most common use-cases is important". -- Dave Ford
I agree, and I will suggest that the next version of the spec (post 1.0.1) explicitly include a use case section. I have argued in the expert group that the topmost goal for JDO 2.0 should be "to encourage the adoption of JDO by evolving it in directions that meet the needs of those interested in using JDO." In particular, JDO should evolve, IMO, in the direction of making it easier to use JDO in today's applications.

11471 Active Members!
[JOIN NOW!](#)



Glad to see JDO 2.0 starts!, Hope it won't be long before it ends

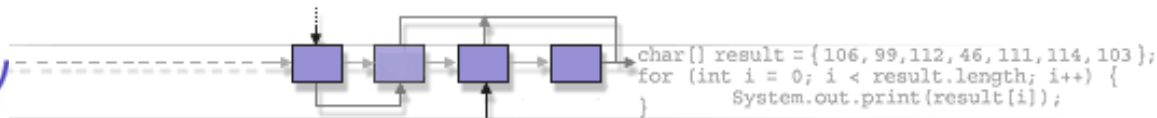
All the dreams come true...My first suggestion will be: define an optional set of features with respect to JDBC datastores. And I believe most implementations will implement this set for the market...

Site sponsored and powered by the Charter Members listed in the [Vendor Directory](#)
Copyright© - Companion Systems 2003. All Rights Reserved. Read our [Privacy Policy](#) for details.

Site sponsored and powered by [FastObjects](#). Copyright © 2002. All Rights Reserved.
[Terms of Use](#) | [Privacy Policy](#)



Java
Community
Process



Quick Links: [What's New](#) [JCP 2](#) [Membership](#) [Spec Lead Guide](#) [java.sun.com](#)

Introduction

JSRs:
Java Specification Requests

What's New

Participation

JCP Procedures

Press & Success

Community Resources

JSR-000012 Java™ Data Objects (JDO) Specification (Maintenance Release)

This is an updated Final Release of this Specification, as described in [Section 4.2](#) of the Java Community ProcessSM Program, version 2.1.

Specification:

- [Download](#)

Maintenance Lead:

- Craig Russell
Sun Microsystems, Inc.

Please send comments to jdo-comments@sun.com.

Reference Implementation and Technology Compatibility Kit:

- [JDO Reference Implementation Source](#)
- [JDO Reference Implementation Binary](#)
- [JDO Technology Compatibility Kit](#)

Change Log:

- [JSR 12 Change Log](#)

See Also

[JSR-000012 Java™ Data Objects Page](#)

[Previous Final Release of JSR-000012 Java™ Data Objects Specification](#)

[JCP Final Releases Page](#)

[Community Process Main Page](#)

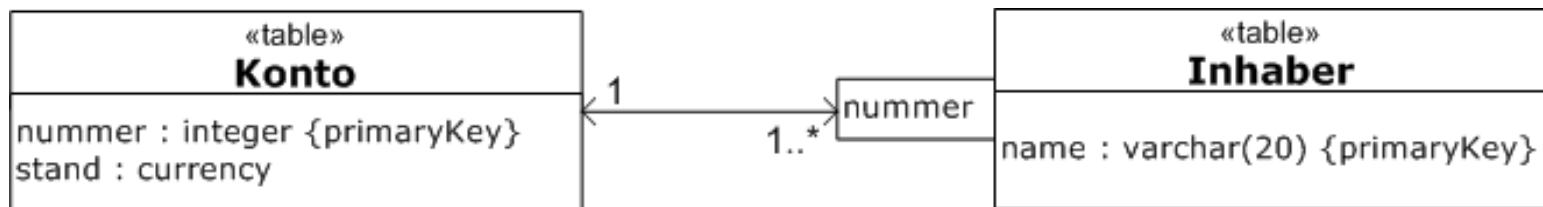
Search

[Introduction](#) | [JSRs](#) | [What's New](#) | [Participation](#)
[JCP Procedures](#) | [Press & Success](#) | [Community Resources](#)

Site sponsored and powered by [Sun Microsystems](#)
Copyright © 1995-2003. All Rights Reserved.
[Terms of Use](#). [Privacy Policy](#).







```
CREATE TABLE konto(nummer INTEGER PRIMARY KEY, stand DECIMAL(5,2) NOT NULL);  
CREATE TABLE inhaber(name VARCHAR(20) NOT NULL, nummer INTEGER NOT NULL,  
PRIMARY KEY(name, nummer));
```

```
INSERT INTO konto VALUES(12345678, 5000);  
INSERT INTO konto VALUES(11111111, 100);
```

```
INSERT INTO inhaber VALUES("John Doe", 12345678);  
INSERT INTO inhaber VALUES("John Doe", 11111111);  
INSERT INTO inhaber VALUES("Barnie Bar", 11111111);
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Bank {
    public static void Überweisung(
        int srcAccount,
        int dstAccount,
        double amount) {
        Connection con = connectDB();
        try {
            con.setAutoCommit(false);
            con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
            Statement stmt = con.createStatement();
            stmt.executeUpdate("UPDATE konto set stand=stand-"+amount+" where
nummer='"+srcAccount+"'");
            stmt.executeUpdate("UPDATE konto set stand=stand-"+amount+" where
nummer='"+dstAccount+"'");
            con.commit();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

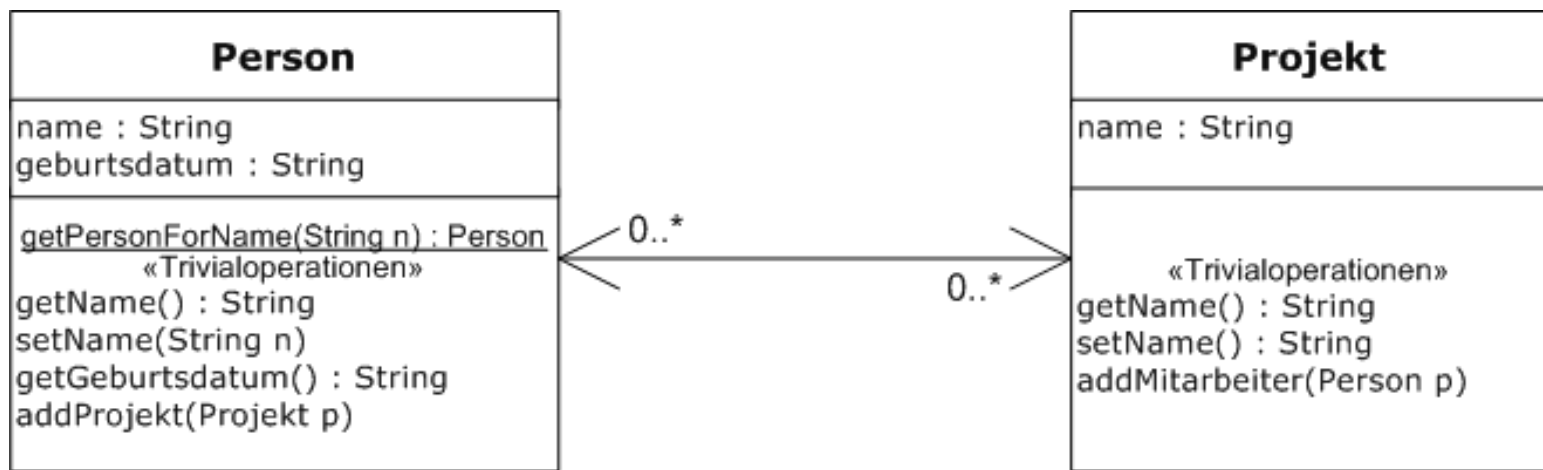
    public static String Vermögensübersicht(String kunde) {
        String result=null;
        try {
            result= "Vermögensübersicht für: "+kunde+"\n";
            result+="-----\n";
            Statement stmt = connectDB().createStatement();
            stmt.executeUpdate("LOCK TABLES konto READ, inhaber READ;");
            ResultSet rs = stmt.executeQuery("SELECT k.nummer, stand FROM konto AS k,
inhaber AS i WHERE i.name='"+kunde+"' AND i.nummer=k.nummer;");
            while(!rs.isLast()) {
                rs.next();
                result+=rs.getInt("nummer")+"\t"+rs.getInt("stand")+"\n";
            }
            result+="-----\n";
            rs = stmt.executeQuery("SELECT SUM(stand) AS s from konto as k,inhaber as i
where i.name='"+kunde+"' and i.nummer=k.nummer;");
            stmt.executeUpdate("UNLOCK TABLES;");
            rs.next();
            result+="Saldo aller Konten: "+rs.getInt("s")+"\n";
        } catch (SQLException e) {
```

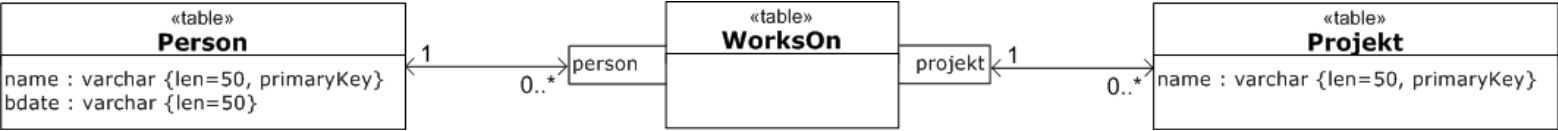
```
        e.printStackTrace();
    }
    return result;
}
private static Connection connectDB() {
    try {
        Class.forName("com.mysql.jdbc.Driver");
    } catch (ClassNotFoundException e) {
        System.err.println("Driver class not found");
        e.printStackTrace();
    }
    Connection con = null;
    try {
        con =
            (Connection) DriverManager.getConnection(
                "jdbc:mysql://localhost/bank/",
                "root",
                "");
    } catch (SQLException e1) {
        System.err.println("Error establishing database connection");
        e1.printStackTrace();
    }
    return con;
}
}
```

```
public class Driver {  
    public static void main(String args[]) {  
        Bank.Überweisung(12345678, 11111111, 500);  
        System.out.println(Bank.Vermögensübersicht("John Doe"));  
    }  
}
```

```
public abstract class TransactionScript {  
    public void run(){  
    }  
}
```

```
public class Driver2 {  
    public static void main(String args[]) {  
        Überweisung ü = new Überweisung(12345678, 11111111, 500);  
        ü.run();  
  
        Vermögensübersicht v = new Vermögensübersicht("John Doe");  
        v.run();  
        System.out.println(v.result);  
    }  
}
```





```
CREATE TABLE Person(  
    name VARCHAR(50) PRIMARY KEY,  
    bdate DATE);
```

```
CREATE TABLE Project(  
    name VARCHAR(50) PRIMARY KEY);
```

```
CREATE TABLE WorksOn(  
    person VARCHAR(50) NOT NULL,  
    project VARCHAR(50) NOT NULL,  
    PRIMARY KEY (person, project));
```

```
ALTER TABLE WorksOn ADD INDEX WO_person_IDX(person);  
ALTER TABLE WorksOn ADD CONSTRAINT WO_person_FK FOREIGN KEY (person)  
REFERENCES Person(name);  
ALTER TABLE WorksOn ADD INDEX WO_project_IDX(project);  
ALTER TABLE WorksOn ADD CONSTRAINT WO_project_FK FOREIGN KEY (project)  
REFERENCES Project(name);
```

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Vector;

public class Person {
    private String name;
    private Vector projekte = new Vector();

    public void setName(String name) {
        Statement stmt = DBConnector.getConnectionStatement();
        try {
            ResultSet rs =
                stmt.executeQuery(
                    "SELECT count(*) FROM Person WHERE name="
                    + this.name
                    + ";"");
            rs.next();
            if (rs.getDouble(1) < 1) {
                System.out.println(
                    "Database entry for person does not exist and will be created");
                stmt.executeUpdate(
                    "INSERT INTO Person (name) VALUES(" + name + ")");
            } else {
                System.out.println(
                    "Database entry for person exists and will be updated");
                stmt.executeUpdate(
                    "UPDATE Person SET name="
                    + name
                    + " where name="
                    + this.name
                    + ";"");
            }
        } catch (SQLException e) {
            System.out.println("Cannot access database");
            e.printStackTrace();
        }
        this.name = name;
    }

    public void setGeburtsdatum(String gebDat) {
        Statement stmt = DBConnector.getConnectionStatement();
        try {
            System.out.println("Database entry (bdate) will be updated");
            stmt.executeUpdate(
                "UPDATE Person SET bdate="
```

```
        + gebDat
        + " WHERE name="
        + this.name
        + "");
    } catch (SQLException e) {
        System.out.println("Cannot access database");
        e.printStackTrace();
    }
}

public String getGeburtsdatum() {
    Statement stmt = DBConnector.getConnectionStatement();
    ResultSet rs;
    String result = "";
    try {
        rs =
            stmt.executeQuery(
                "SELECT bdate FROM Person WHERE name=" + name + "");
        rs.next();
        result = rs.getString(1);
    } catch (SQLException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    return result;
}

public String getName() {
    return name;
}

public void addProjekt(Projekt p) {
    Statement stmt = DBConnector.getConnectionStatement();
    try {
        stmt.executeUpdate(
            "INSERT INTO WorksOn VALUES("
                + this.name
                + ", "
                + p.getName()
                + ")");
    } catch (SQLException e) {
        System.out.println("Cannot access database");
        e.printStackTrace();
    }
}

public Vector getAllProjekt() {
    Vector projekte = new Vector();

    return projekte;
}
```

```
}
public static Person getPersonForName(String name) {
    Statement stmt = DBConnector.getConnectionStatement();
    ResultSet rs;
    Person p = new Person();
    try {
        rs =
            stmt.executeQuery(
                "SELECT bdate FROM Person WHERE name='" + name + "';");
        rs.next();
        p.name = name;
    } catch (SQLException e) {
        System.out.println("Cannot access database");
        e.printStackTrace();
    }
    return p;
}
public String toString() {
    return "(name: "+name+", bdate: "+this.getGeburtsdatum()+")";
}
}
```

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Vector;

public class Projekt {
    private String name;
    private Vector mitarbeiter = new Vector();

    public void setName(String name) {
        Statement stmt = DBConnector.getConnectionStatement();
        try {
            ResultSet rs =
                stmt.executeQuery(
                    "SELECT count(*) FROM Project WHERE name='"
                        + this.name
                        + "';");
            rs.next();
            if (rs.getDouble(1) < 1) {
                System.out.println(
                    "Database entry for project does not exist and will be created");
                stmt.executeUpdate(
                    "INSERT INTO Project (name) VALUES('" + name + "')");
            } else {
                System.out.println(
                    "Database entry for project exists and will be updated");
                stmt.executeUpdate(
                    "UPDATE Project SET name='"
                        + name
                        + "' where name='"
                        + this.name
                        + "';");
            }
        } catch (SQLException e) {
            System.out.println("Cannot access database");
            e.printStackTrace();
        }
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void addMitarbeiter(Person p) {
        Statement stmt = DBConnector.getConnectionStatement();
        try {
```

```
        stmt.executeUpdate(
            "INSERT INTO WorksOn VALUES("
                + p.getName()
                + ", "
                + this.name
                + ");");
    } catch (SQLException e) {
        System.out.println("Cannot access database");
        e.printStackTrace();
    }
}

public Vector getAllMitarbeiter() {
    Vector mitarbeiter = new Vector();
    Statement stmt = DBConnector.getConnectionStatement();
    Person p;
    try {
        ResultSet rs = stmt.executeQuery("SELECT name FROM Person as p, WorksOn as w
WHERE w.project='"+this.name+"' and w.person=p.name;");
        while (!rs.isLast()) {
            rs.next();
            p = Person.getPersonForName(rs.getString(1));
            mitarbeiter.add(p);
        }
    } catch (SQLException e) {
        System.out.println("Cannot access database");
        e.printStackTrace();
    }
    return mitarbeiter;
}

}
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class DBConnector {
    public static Statement getConnectedStatement() {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Driver class not found");
            e.printStackTrace();
        }
        Connection con = null;

        try {
            con =
                (Connection) DriverManager.getConnection(
                    "jdbc:mysql://dbServerMachine/mcjtest/",
                    "user",
                    "thePassword");
            return (con.createStatement());
        } catch (SQLException e1) {
            System.err.println("Error establishing database connection");
            e1.printStackTrace();
        }
        //never gets here
        return null;
    }
}
```



```
public class TestDriver {  
    public static void main(String args[]) {  
        Person pers1 = new Person();  
        pers1.setName("Max Mustermann");  
        pers1.setName("XX");  
        pers1.setName("Max Mustermann");  
        pers1.setGeburtsdatum("1970-11-12");  
  
        Person pers2 = new Person();  
        pers2.setName("John Doe");  
  
        Projekt prj1 = new Projekt();  
        prj1.setName("Reorganisation");  
  
        pers1.addProjekt(prj1);  
        prj1.addMitarbeiter(pers2);  
  
        System.out.println(prj1.getAllMitarbeiter());  
    }  
}
```

▲ **Home**

▶ **Dialog**

▲ **Fragen, Anregungen, Wünsche? Die Feedback-Seite!**

Hyperlinks

- ☐ Ungültiger Hyperlink ...
- ☐ Überflüssiger Hyperlink ...
- ☐ fehlender hyperlink ...

... von Seite:

... nach Seite:

Bemerkungen:

Seiteninhalt

- ☐ Nicht mehr aktueller Inhalt ...
- ☐ Inkorrektter Inhalt ...
- ☐ Irreführender Inhalt ...

... auf Seite:

Bemerkung:

Anregungen, Wünsche, Fragen

Bitte tragen Sie weitere Anregungen, Wünsche und Fragen hier ein:

Und nun...

Ihr Name:

Ihre Email-Adresse:

ab die Post!

Eingaben löschen

Service provided by [Mario Jeckle](#)

Generated: 2003-12-24T17:12:22+01:00

▶ [Feedback](#)

▶ [SiteMap](#)

▶ [This page's original location: http://www.jeckle.de/feedback.html](#)

▶ [RDF description for this page](#)

▲ Home

- ▼ [Unified Modeling Language \(UML\)](#)
- ▼ [eXtensible Markup Language \(XML\)](#)
- ▼ [XML Metadata Interchange \(XMI\)](#)
- ▼ [Web Services](#)
- ▼ [Vorträge und Publikationen](#)
- ▼ [Vorlesungen](#)
- ▼ [Studien- und Diplomarbeiten](#)
- ▼ [Internet Search Engines / Suchmaschinen](#)
- ▼ [Mersennesche Primzahlen](#)
- ▼ [Web Services Workshop WS-RSD'02](#)
- ▼ [Free Stuff --- Software & Downloads](#)
- ▼ [RSS-Newsfeeds](#)

- ▶ [International Conference on Web Services'03-Europe](#)
- ▶ [Web Services @ Berliner XML-Tage](#)
- ▶ [XML-Strategie](#)
- ▶ [XML Acronym Demystifier Project](#)
- ▶ [GOOAL.net](#)
- ▶ [feedback](#)
- ▶ [Rotkreuz Mitgliederverwaltung](#)
- ▶ [Suchen ...](#)
- ▶ [Feedback](#)
- ▶ [Dialog ...](#)
- ▶ [Über diese Seiten ...](#)
- ▶ [Was gibt's hier Neues?](#)
- ▶ [RSS Newsfeed](#)
- ▶ [Mario Jeckle](#)
- ▶ [Mein PGP public key für Mailverschlüsselung, etc.](#)



Unified Modeling Language (UML)

- [UML Resource Center](#)
- [UML Links](#)
 - [Generelles](#)
 - [Tutorials](#)
 - [Frequently Asked Questions \(FAQ\) und Mailinglisten](#)
 - [UML @ OMG](#)
 - [Austauschformate](#)
 - [Werkzeugbezogenes ...](#)
 - [Objektorientierung allgemein](#)
 - [UML Autoren](#)
 - [UML auf gut Deutsch](#)(Vorschlag einer Gruppe deutschsprachiger Autoren zur einheitlichen deutschen Übersetzung der UML-Begriffe)
- [UML Bücher](#)
- [UML Official Specification Documents / Offizielle UML Spezifikation](#)
 - [General Information](#)
 - [UML Profiles und sonstige UML-bezogene Aktivitäten](#)
 - [UML v2.0](#)
 - [UML v1.5](#)
 - [UML v1.4](#)
 - [UML v1.3](#)
 - [UML v1.2](#)
 - [UML v1.1](#)
 - [Unterschiede zwischen den UML-Versionen](#)
- [UML Online Publications / Netz Ressourcen zur UML](#)
- [UML Tools](#)



eXtensible Markup Language (XML)

- [XML Resource Center](#)
 - [Was ist die XML?](#)
 - [Links](#)
 - [Parser](#)
 - [XML-Editoren](#)
 - [XML \(Einsteiger-\)Informationen & Tutorials](#)

- [Mailinglisten und User Groups](#)
- [Konferenzen](#)
- [FAQs und Antworten darauf](#)
- [Vorlesung XML](#)
- [Entwurf von XML-Sprachen](#)
- [XML und Sicherheit \(Digitale Signatur, Verschlüsselung, ...\)](#)
- [Eine Übersicht existierender XML-Sprachen](#)
 - [Channel Description Format \(CDF\)](#)
 - [Chemical Markup Language \(CML\)](#)
 - [Customer Profile Exchange \(CPEX\)](#)
 - [Mathematical Markup Language \(MathML\)](#)
 - [Querying XML \(XQuery\)](#)
 - [RDF Site Summary \(RSS\)](#)
 - [Resource Description Framework \(RDF\)](#)
 - [Simple Markup Language \(SML\) und verwandte Ansätze \(TMML, Minimal XML, YAML, ...\)](#)
 - [Voice ML](#)
 - [Wireless Markup Language \(WML\)](#)
 - [XML Hypertext Markup Language \(XHTML\)](#)
 - [XML Linking Language \(XLink\)](#)
 - [XML Path Language \(XPath\)](#)
 - [XML Pointer Language \(XPointer\)](#)
 - [XML Protocols \(XML-RPC, SOAP, W3C's XML-Protocols, ...\)](#)
 - [XML-Schema \(XSD, DCD, DT4DTD, Schematron, RELAX, ...\)](#)
 - [Projekt zur Übersetzung der XML-Schema-Spezifikation ins Deutsche](#)
 - [XSD-DE Chat vom 2001-08-16](#)
 - [XSD-DE Chat vom 2001-09-11](#)
 - [XSD-DE Chat vom 2001-10-08](#)
 - [XML Stylesheet Language \(XSL\)](#)
 - [XSL Transformations \(XSLT\)](#)
 - [Konversion nativer Formate nach XML](#)
- [Vorlesung eXtensible Markup Language](#)



XML Metadata Interchange (XMI)

- [XMI Resource Center](#)
- [Beispiele](#)
 - [Einfaches UML-Diagramm in XMI v1.0, v1.1 und als Rational Rose Export](#)
 - [Umfangreicheres UML-Diagramm in XMI v1.0](#)
 - [Generierung einer eigenen XMI-Sprache](#)
 - [Nutzung der MOF-DTD: UML v1.4 Metamodell in XMI v1.1](#)
 - [Generierung eines XSD-Schemas aus einem UML-Klassendiagramm unter Nutzung von XMI\[UML\]](#)
- [XMI News](#)
- [Werkzeuge mit XMI-Support](#)
- [Wozu ein Metadaten-Austauschformat?](#)
- [Ziele von XMI](#)
- [Hintergründe zur Entwicklung von XMI](#)
- [Offizielle Spezifikationsdokumente](#)
- [XMI Production for XML Schema](#)
- [Präsentationen, Vorträge und Veröffentlichungen](#)
- [Links](#)
- [Entwurf von XML-Sprachen mit XMI](#)
- [Modellaustausch mit dem XML Metadata Interchange Format](#)



Vorträge und Publikationen

Vorträge und Publikationen

Abstracts und Kurzfassungen



Vorlesungen

- [Systemarchitekturen](#)
 - [Vorlesungsinhalt](#)
 - [Mailingliste zur Vorlesung](#)
 - [Literatur](#)
 - [Aktuelles XML](#)
- [Standardisierung](#)
 - [Abstract](#)
 - [Ziel der Veranstaltung](#)
 - [Durchgeführtes Projekt](#)
 - [Links](#)
 - [Mailingliste zur Vorlesung](#)
 - [Aktuelles XML](#)
- [DB-Anwendungen](#)

- [Vorlesungsinhalt](#)
- [Mailingliste zur Vorlesung](#)
- [Literatur](#)
- [Links](#)
- [Aktuelles](#) **XML**
- [e-Business Engineering](#)
 - [Vorlesungsinhalt](#)
 - [Mailingliste zur Vorlesung](#)
 - [FAQs zur Vorlesung](#)
 - [Aktuelles](#) **XML**
- [Datenbanken](#)
 - [Vorlesungsinhalt](#)
 - [Mailingliste zur Vorlesung](#)
 - [Übungsaufgaben](#)
 - [FAQs zur Vorlesung](#)
 - [Aktuelles](#) **XML**
- [eXtensible Markup Language](#)
 - [Teleteaching](#)
 - [Vorlesungsinhalt](#)
 - [Gliederung](#)
 - [Termine](#)
 - [FAQs zur Vorlesung](#)
 - [Anmeldung zum Praktikum im WS2002/03](#)
 - [Übungsaufgaben:](#)
 - [1. Übungsaufgabe](#)
 - [2. Übungsaufgabe](#)
 - [Hinweise, Anmerkungen und FAQs zu dieser Übung](#)
 - [3. Übungsaufgabe](#)
 - [Hinweise, Anmerkungen und FAQs zu dieser Übung](#)
 - [4. Übungsaufgabe](#)
 - [Hinweise, Anmerkungen und FAQs zu dieser Übung](#)
 - [5. Übungsaufgabe](#)
 - [Hinweise, Anmerkungen und FAQs zu dieser Übung](#)
- [Alte Klausuren](#)
- [Script](#)
- [Hinweise zum Script](#)
- [Empfohlene Literatur](#)
- [Archiv: Praktikumsaufgaben und -Lösungen aus dem WS2001/02](#)

• [Java Threads](#)

- [Inhaltsübersicht](#)
- [Script](#)
- [Klausur vom Sommersemester 2002](#)
- [Literatur](#)
- [Internetressourcen zum Thema](#)

• [Java](#)

- [Script](#)
- [Gliederung](#)
- [Abstract](#)

• [Systemnahe Software](#)

• [Software Engineering I](#)

- [Hands on Java](#)
- [Objektorientierung](#)
- [Applikationserstellung](#)
- [Anwendungsaspekte](#)

• [Chat zur Vorlesung](#)



Studien- und Diplomarbeiten

- [Mögliche noch zu vergebende Studien- und Diplomarbeitsthemen](#)
- [Gegenwärtig laufende Arbeiten](#)
- [Abgeschlossene Arbeiten](#)



Web Services

- [Web Services -- eine technisch orientierte Einführung](#)
- [Sicherheitsaspekte XML-basierter Web-Services](#)
- [Web Service Workshop WS-RSD'02](#)
- [Web Services @ Berliner XML-Tage](#)



GOOAL.net

Deutsche Mailingliste rund um die Objekttechnologie

[direkt zu GOOAL.net ...](#)



Internet Search Engines / Suchmaschinen

- [Deutsche Suchmaschinen](#)
- [Meta-Suchmaschinen](#)
- [Internationale Suchmaschinen](#)
- [E-Mail Finder](#)
- [Dictionaries und Übersetzungshilfen](#)
- [File Finder](#)
- [Books & Journals](#)
- [Page Registrierung](#)

▲ Mersennesche Primzahlen

- [Mersennesche Primzahlen](#)
- [SETI @ Home](#)
- [Rätsel der Proteinfaltung](#)

▲ Web Services Workshop WS-RSD'02

- [Workshop Homepage](#)
- [Call for Papers](#)
- [Workshop Program, Schedule, and Presentation Slides Workshop Program, Schedule, and Presentation Slides](#)

▲ Free Stuff --- Software & Downloads

- [XInclude Task for Ant](#)
- [Java version of Linpack benchmark](#) **New**

▲ RSS-Newsfeeds

- [jeckle.de Newsfeed \(HTML-Version\)](#)
- [DB-Anwendungen Newsfeed \(HTML-Version\)](#)
- [e-Business Engineering Newsfeed \(HTML-Version\)](#)
- [Datenbanken Newsfeed \(HTML-Version\)](#)
- [Mehr Informationen zu RSS](#)



Feedback

[direkt zu Seite ...](#)



Rotkreuz Mitgliederverwaltung

[direkt zu Seite ...](#)



Dialog

[direkt zu Seite ...](#)



Über diese Seiten ...

[direkt zu Seite ...](#)



Suchen ...

[direkt zu Seite ...](#)



Mario Jeckle ...

[direkt zu Seite ...](#)

[PGP public key](#)

Service provided by [Mario Jeckle](#)

Generated: 2004-01-01T17:02:27+01:00

► [Feedback](#) ► [SiteMap](#)

► [This page's original location: http://www.jeckle.de/sitemap.html](#)

► [RDF description for this page](#)