

SQL-J

SQL-J

- SQL-J ist ein Standard aus drei Teilen, welcher die Verwendung von Java-Technologien *in* einer Datenbank regelt:
 1. Gebrauch von Java für SQL-Funktionen und Prozeduren (ANSI 331.1 1999 SQL-J Part 1)
 2. Gebrauch von Java-Klassen als SQL-Datentypen (ANSI 331.2 2000 SQL-J Part 2)
 3. Einbettung von SQL-Befehlen in Java-Programme (ISO 9075-10 2000 SQL/OLB)

Achtung: SQL/OLB hat nichts mit JDBC zu tun. SQL/OLB ermöglicht es, analog zum früheren embedded SQL for C, statische SQL-Befehle in Java einzubinden. Der Vorteil liegt in der besseren Überprüfbarkeit der SQL-Befehle zur Kompilationszeit und eventuell in einer besseren Performance zur Laufzeit.

Warum SQL-J ?

- Drang der Datenbanktechnologie nach :
 - Verwendung objektorientierter Prinzipien
 - Vermeidung der starken Dekomposition beim Normalisieren
 - Vereinheitlichung der proprietären Sprachen für Server-Sided-Programming in Triggern und Stored Procedures
 - Beibehaltung und Nutzung so wunderbarer Dinge wie :
 - SQL (einfach, optimiert, gelebter Standard)
 - Java (Allzweck-Sprache, robust, Leading Technology)
- Einen Standard als Leitlinie für Produkthersteller und Applikationsentwickler bereitstellen, der SQL und Java verbindet.

Das Halten von ganzen Objekten in der Datenbank, statt einfacher Werte, bietet verschiedene Vorteile. Beispielsweise ist die Adresse einer Person zwar strukturiert in Strasse, PLZ und Ort, wird aber immer als Ganzes in einer Applikation verwendet. Der Transfer von und zur Applikation kann als ganzes Objekt und nicht in Form einzelner Felder durchgeführt werden. Das vereinfacht und generalisiert einerseits die Programmierung der Applikation, und fördert andererseits die Kompaktheit der Datenbank, indem nur ein einziges Tabellenfeld 'Adresse' benötigt wird. Unter Umständen können sogar mehrere Adressen (derselben Person beispielsweise) in einziges Tabellenfeld gepackt werden, wenn der Datentyp des entsprechenden Tabellenfeldes eine 'Liste von Adressen' ist.

Die Vererbung ist ein sehr hilfreiches Konstrukt der OO-Methodik: Beispielsweise können damit unterschiedliche Adresstypen in einem einzigen Feld einer SQL-J Tabelle untergebracht werden.

Kapselung und Zugriffsmethoden auf einzelne Attribute eines Objektes (bereits in der Datenbank und nicht erst in der Applikation) helfen, fehler- und widerspruchsfreie Applikationen zu entwickeln.

Ein besonderes Augenmerk ist auf die Optimierung von SQL-Abfrage- und –Modifikationsbefehle zu richten. Tendenziell entstehen hier mehr Probleme mit SQL-J als mit reinem SQL.

Gewisse Design-Fragen treten neu in Erscheinung, beispielsweise:

- Was ist die Semantik von statischen und transienten Attributen einer Klassen bezüglich Speicherung in einer DB?
- Können sich verschiedene Objekte in der Datenbank gegenseitig referenzieren, wie bei gewöhnlichen Java-Programmen?
- Welche Daten werden als Objekte abgelegt, welche als reine Werte im klassischen SQL-Sinn?
- Was passiert mit bereits bestehenden Objekten, wenn sich eine Klassen-Definition ändert?

Anwendungsbeispiele

<i>Anwendungsbeispiel</i>	<i>Beispiel Java-Klasse</i>
• Array von Messwerten • Link auf Webseite • Klassenhierarchie von geometrischen Figuren • XML-Dokument • Verschlüsseltes Objekt • Signiertes Objekt • Komprimiertes Objekt • Texte in div. Sprachen • Eigene Datentypen • Strategy-Pattern	<pre>java.util.ArrayList java.net.URL MyShape implements java.awt.Shape, Serializable org.jdom.Document javax.crypto.SealedObject java.security.SignedObject MyCompressedData MyMultiLanguageText DateIntervall, Regexp</pre>

Alle speicherbaren Klassen müssen das Interface `java.io.Serializable` implementieren.

Interfaces selber können nicht als Datentypen für ein Tabellenfeld verwendet werden. Arrays in der Schreibweise `Klasse variable[]` können ebenfalls nicht als Datentypen verwendet werden. Ein Workaround ist leicht zu bewerkstelligen, indem Variablen des gewünschten Interfaces oder des gewünschten Array-Typs in eine gewöhnlichen Klasse als Member eingebaut werden.

Bei vielen Anwendungsbeispielen mit Javaklassen könnte der zugrundeliegende Datentyp auch aus dem SQL-Umfang "rekrutiert" werden. Interessant ist in diesem Fall vor allem die zusätzliche Funktionalität in Form von Objektmethoden, welche in SQL direkt verwendet werden kann.

Zum letzten Anwendungsbeispiel: Das Strategy-Pattern ist definiert durch eine Basisklasse welche eine abstrakte `doit()`-Methode enthält. Die Methode ist dann in den Unterklassen implementiert. Ein Client ruft einfach die `doit()`-Methode auf, die er von irgendjemandem bekommt. Damit bleibt ihm der genaue Algorithmus für eine Aufgabe verborgen, resp. ist leicht austauschbar. Als Beispiel könnte beispielsweise eine `similar`-Funktion für Strings erhalten: String-Ähnlichkeiten könnten mit regulären Ausdrücken, mit Soundex-Algorithmen, mit Differenz-Verfahren etc. bestimmt werden.

Warum *nicht* SQL-J ?

- Klassischerweise ist eine Datenbank unabhängig von der Programmiersprache, welche sie benützt.
- Durch SQL-J wird eine Java-Abhängigkeit eingeführt.
- Die Verwendung von Java kann auf die Serverseite beschränkt werden:
 - Java wird nur für die Programmierung von instanzunabhängigen Methoden (Stored Procedures, Triggers, allgemeinen Funktionen) verwendet.
 - Beim Zugriff auf Objekte mit `select` werden immer Methoden aufgerufen, welche SQL-Basistypen zurückliefern.
 - Java-Objekte werden innerhalb von `insert`-Befehlen instanziiert.

Welche Objekte eignen sich für SQL-J ?

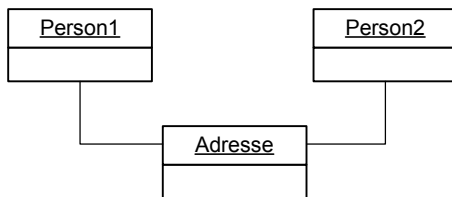
- Isolierte Objekte oder Objekt-Kompositionen
- Klassen mit Vererbungshierarchien
- Datenobjekte, nicht Funktionsobjekte
- Datenobjekte, welche spezielle Zugriffsfunktionen oder Operatoren benutzen
- Objekte mit wenig SQL-Abfragen auf Attribute oder Methoden.

int	varchar	date	MyClass

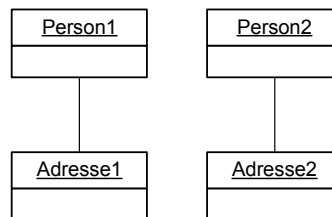
Objekte in SQL-J sind Non-Shareable !

- Im Gegensatz zu 100% objektorientierten Datenbanken ist ein Objekt in SQL-J immer in einem Tabellenfeld abgelegt.
- Objekte sind ausserhalb des eigenen Tabellenfeldes nicht shareable.

nicht möglich

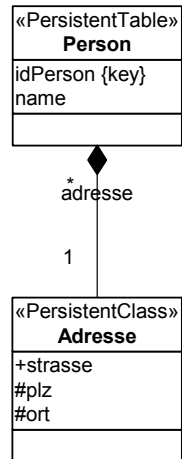
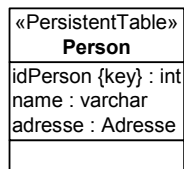


möglich



UML-Design

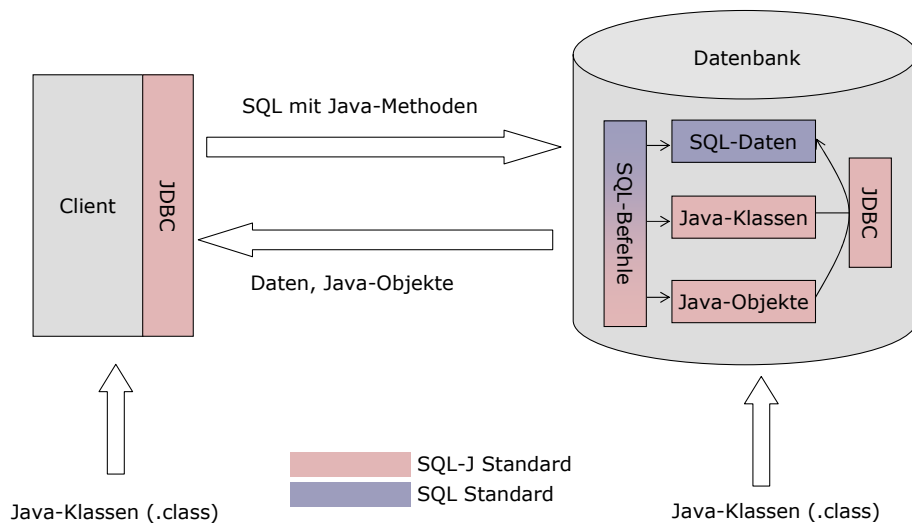
- Auf der Ebene eines konzeptionellen Modells (Analyse) können Klassen explizit dargestellt werden oder nur als Datentyp für Attribute in Erscheinung treten.



Im Gegensatz zu 100% objektorientierten Datenbanken ist ein Objekt in SQL-J immer in einem Tabellenfeld abgelegt.

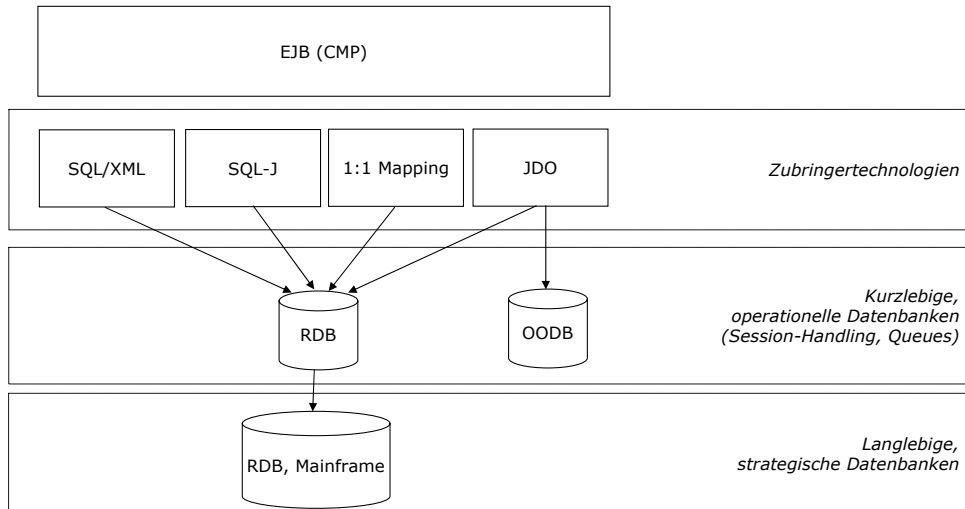
Ausserst wichtig für die Datenmodellierung ist, dass Objekte aussserhalb des eigenen Tabellenfeldes nicht shareable sind!

Anwendungsarchitektur von SQL-J



Alle grossen Vertreter auf dem Datenbanmarkt bieten SQL-J Lösungen an.

Einbettung in andere Architekturen



EJB gilt generell als problematisch bezüglich Performance mit einem 1:1 Mapping zwischen Entity-Beans und Datensätzen. Die EJB-Entwickler experimentieren deshalb mit verschiedenen Zubringertechnologien, welche es ermöglichen, komplexe Businessobjekt bereits in der Datenbank zu halten, resp. in der Datenbank zu generieren, statt im Applikationsserver. SQL-J ist eine dieser Zubringertechnologien, nebst der direkten Verwendung für Standalone-Applikationen.

Neue Technologien sind insbesondere gut einsetzbar, wenn sie mit operationellen Daten arbeiten. Der Lebenszyklus der Daten ist damit sicher kleiner als derjenige der Technologie. Es ergeben sich keine Datenmigrationsprobleme.

Beispiel Sybase ASA

- Ausgehend vom Teil 2 des SQL-J Standards soll nun die Verwendung von Java als Datenbanksprache, in Kombination mit SQL, erläutert werden.
- Alle Beispiele orientieren sich syntaktisch an Sybase Adaptive Server Anywhere 8.0. Wo *konzeptionell* starke Abweichungen zum SQL-J Standard bestehen, wird dies notiert.
- Alle bekannten Produkte (DB2, Oracle, Sybase, Cloudscape, Informix usw., jedoch nicht SQL Server) bieten SQL-J Technologien an.

Sybase ASA hat eine einfache und stabile Administration für den Umgang mit Java-Klassen. Klassen können mit einem install-Befehl installiert werden und sie ab diesem Zeitpunkt als Datentypen in der SQL-Umgebung verfügbar. Besonders hervorzuheben ist, dass eine geänderte Klassendefinition über ein bestehende geladen werden kann, und Sybase versucht, die bestehenden Objekte entsprechend der neuen Definition anzupassen.

Das Produkt Cloudscape ist insofern hervorzuheben, weil es eine rein in Java geschriebene Datenbank von IBM ist. Sie wird von Sun in der J2EE Referenz-Implementation verwendet. Bezüglich SQL-J geht Cloudscape einen schlanken Weg: Alle im CLASSPATH verfügbaren Klassen können in der Datenbank sofort genutzt werden. Der SQL-J Standard sieht einen etwas umständlicheren Weg vor, indem jede als Datentyp zu verwendende Klasse explizit import werden muss. Zwei Gründe sind wahrscheinlich ausschlaggebend für den etwas komplizierten Weg im Standard:

1. Die vorherrschenden Produkte auf dem Datenbank-Markt sind nicht java-basiert. Sie bieten zwar SQL-J ebenfalls an, aber auf der Basis einer nachträglich in die Datenbank eingebauten Java Virtual Machine. Für sie ist Java-Technologie eine Art Add-On. Nicht alle Kunden wünschen ausserdem Java in der Datenbank. Eine implizite und damit starke Verknüpfung mit einer externen Java-Umgebung ist daher gar nicht gewünscht.
2. Mit dem expliziten Import (create type) ist der Entscheid, was persistente und was nicht-persistente Klassen sind sowie das Versionen-Management zwar aufwendiger aber kontrollierbarer.

Weil Cloudscape eine reine Java-Applikation ist, können sämtliche Java-Klassen (insbesondere auch Threads, IO-Klassen, Netzwerk-Klassen) in Server-Sided Java-Programmen genutzt werden. Als Kommunikationsprotokoll zwischen Client und Datenbank wird in Cloudscape 4.0 RMI verwendet (innerhalb der JDBC-Klassen). Für den Zugriff über Firewalls hinweg ist das nachteilig, in einer Multitier-Applikation, wo die Datenbank mit einem lokalen Webserver kommuniziert spielt es eine untergeordnete Rolle.

Beispielklasse

```
public class Adresse implements java.io.Serializable
{
    public String strasse;
    protected String plz;
    protected String ort;
    ...

    public Adresse( String strasse, String plz, String
    ort ) {}
    public String getPlz() {};
    public String getOrt() {};
    ...
}
```

Damit von einer Klasse Objekte in die Datenbank aufgenommen werden können, muss sie serialisierbar sein, d.h. ihre Attribute müssen in einen Datenstrom geschrieben werden können. Die Java Basistypen (int, float, char, byte und String) sind immer serialisierbar. Viele weitere Klassen aus dem Standard-Sprachumfang sind ebenfalls serialisierbar, beispielsweise `java.util.Date` und `java.util.Vector`. Um anzugeben, dass eine eigene Klasse serialisierbar ist, muss diese das Interface `java.io.Serializable` implementieren. Der Java-Serialisierungsmechanismus arbeitet rekursiv alle Komponentenklassen ab, welche serialisierbar sind. Ist ein Attribut als `transient` gekennzeichnet, wird es nicht serialisiert.

Im SQL-J Standard existiert noch eine weitere Möglichkeit um Objekte in die Datenbank abzubilden. Die Klasse der Datenobjekte muss das Interface `java.sql.SQLData` implementieren. Dieses definiert drei Methoden, `public void readSQL (SQLInput stream, String typeName)`, `public void writeSQL (SQLOutput stream)` und `public String getSQLTypeName()`. Mit `readSQL()` liest JDBC Daten von der DB in die Applikation ein, mit `writeSQL()` schreibt JDBC Daten von der Applikation in die Datenbank. Mit `getSQLTypeName` wird dem JDBC-Treiber mitgeteilt, in welchen SQL-Datentyp er das Datenobjekt abbilden soll. Der Typname ist beliebig. Der Applikationsentwickler muss die drei Methoden implementieren. Die `readSQL()` - und die `writeSQL()` - Methode müssen korrespondierende Aufrufe enthalten, damit keine Fehler entstehen. Anschliessend werden den JDBC-Aufrufen nur noch die eigentlichen Datenobjekte übergeben. Der JDBC-Treiber ruft selber die beiden Methoden `writeSQL()` und `readSQL()` auf.

Das Arbeiten mit `Serializable` ist praktischer, das Arbeiten mit `SQLData` ermöglicht mehr Flexibilität beim Speichern und Laden von Attributen der Datenobjekte.

Tabellendefinition

Java Klasse neu importieren

```
install java new from file 'Adresse.class'
```

Wenn die Klasse bereits einmal geladen wurde

```
install java update from file 'Adresse.class'
```

Tabelle definieren

```
create table Person (  
    idPerson integer default autoincrement,  
    adresse Adresse,  
    ...  
)
```

Die verlangte Klasse muss zuerst installiert werden. Zu beachten ist, dass die Applikation und die Datenbank korrespondierende Klassendeklarationen haben müssen. In Sybase, wie auch in anderen Datenbanken, besteht die Möglichkeit, die Klassendeklaration ebenfalls in der Datenbank abzulegen und sowohl den Datenbank-Programmen wie den Applikationsprogrammen zentral zur Verfügung zu stellen.

Mit dem Importieren wird der gesamte Klassen-Code in der Datenbank gespeichert. Er wird von dort bei Bedarf über eine Sybase-eigene JVM (Java Virtual Machine) abgearbeitet. Der Import-Befehl wird in einem SQL-Frontend eingegeben, wie alle anderen SQL-Befehle aber serverseitig durchgeführt. Die zu importierenden Java-Klassen müssen also im File-System des Servers abgelegt sein.

Nebenbemerkung zur Performance: Das erstmalige Verwenden von Java-Objekten, beispielsweise indem eine Objektmethode aufgerufen wird, bewirkt das Laden der Java Virtual Machine in den Datenbankserver. Dies verzögert den Aufruf um etwa 1-2 Sekunden.

Typ-Erzeugung im SQL-J Standard

```
create type addr external name 'address_classes_jar:Address'
language java
as (
    street_attr varchar(50) external name 'street',
    zip_attr char(10) external name 'zip'
)
static method rec_width ( ) returns integer
external variable name 'recommendedWidth',
method addr ( ) returns addr external name 'Address',
method addr (s_parm varchar(50), z_parm char(10)) returns addr
external name 'Address',
method to_string ( ) returns varchar(255) external name 'toString',
method remove_leading_blanks ( ) returns addr self as result
external name 'removeLeadingBlanks',
static method contiguous (A1 addr, A2 addr) returns char(3)
external name 'contiguous';
```

Der Standard sieht umfangreiche Deklarationen beim Laden einer Java-Klasse vor, wie obiges Beispiel zeigt. Offenbar die meisten Produkte schenken sich jedoch dieses komplizierte (dafür flexible) Vorgehen und übernehmen die Objekte unter Verwendung des Java-Serialisierungsmechanismus in die Datenbank. Der Zugriff auf Objekte passiert immer durch vorgängige Deserialisierung und Bearbeitung in einer internen Java Virtual Machine.

Speichern von Objekten

```
import com.sybase.jdbc2.jdbc.*;

...
Properties p = new Properties();
p.put( "SERVICENAME", "mydatabase" );
p.put( "USER", "dba" );
p.put( "PASSWORD", "sql" );
SybDriver d = new SybDriver();
DriverManager.registerDriver(d);
con = DriverManager.getConnection(
    "jdbc:sybase:Tds:localhost:2638", p );
...
```

Es wird ein JDBC-Driver Objekt geladen und anschliessend eine Connection erstellt, über welche SQL-Statements abgesetzt werden können.

Speichern von Objekten

```
...
Adresse a = new Adresse( "Auweg 3", "3253", "Neuenegg" );
String qstr = "insert into Person ( adresse )
              values ( ? )";
PreparedStatement stmt = con.prepareStatement( qstr );
stmt.setObject( 1, a );
stmt.executeUpdate();
...
```

Das Adress-Objekt wird im Client erzeugt und anschliessend auf sehr generische Art, mit der Methode `setObject()`, an die Datenbank übergeben.

Lesen von Objekten

```
...
con = DriverManager.getConnection( ... );
...

String qstr = "select adresse from Person ";
PreparedStatement stmt = con.prepareStatement( qstr );
ResultSet rs = stmt.executeQuery();
while ( rs.next() ) {
    Adresse a = (Adresse) rs.getObject( 1 );
    ...
}
```

Das Lesen von Adress-Objekten erfolgt über die generische Methode `getObject()`.
Deren Rückgabewert muss in `Adresse` konvertiert werden.

Erzeugen von Objekten

- Java-Objekte können im Client erzeugt und mit JDBC als Parameter eines insert-Befehles an die Datenbank übergeben werden.
- Java-Objekte können aber auch innerhalb des insert-Befehles und damit *datenbankseitig* erzeugt werden.

```
insert into Person ( adresse )  
values ( new Adresse('Auweg 3', '3253', 'Neuenegg' ) )
```

- Erzeugung im insert-Befehl
 - für die Initialisierung von Stammdaten
 - wenn der Konstruktor auf DB-seitige Ressourcen zugreift

Wenn ein Objekt innerhalb des insert-Befehles erzeugt wird, heisst das gleichzeitig, es wird durch die Java Virtual Machine (JVM) im Datenbankserver erzeugt und serialisiert, nicht durch die JVM im Client. Dies kann von Vorteil sein, wenn sich Ressourcen zur Konstruktion des Objektes ebenfalls auf dem Datenbankserver befinden oder durch diesen angezapft werden, beispielsweise andere Datenbanken, Internet-Verbindungen, Intranet-Verbindungen usw. Die serverseitige Konstruktion ist damit unter Umständen schneller oder aufgrund von Zugriffsberechtigungen, Firewall-Restriktionen überhaupt nur dort möglich.

Die Konstruktion auf der Server-Seite kann ebenfalls beim Aufbau von Stammdaten, Hilfsinformation etc. von Vorteil sein.

Abfragen mit select

- Abfrage des ganzen Objektes

```
select adresse from Person
```

- Abfragen von Attributen

```
select idPerson, adresse.strasse from Person
```

Abfragen mit Methodenaufrufen

```
select adresse.toString() from Person
```

```
select adresse.getOrt() from Person  
where adresse.getPlz() = '3006'
```

- Die Angaben `public`, `protected`, `private` gelten auch in SQL.
- Modifizierende Methodenaufruf verändern nur eine Kopie des Objektes!

In einem interaktiven SQL-Frontend wird anstelle des Objektes das Resultat von `toString()` zurückgegeben.

Im SQL-J Standard kann ein Java-Datentyp so definiert werden, dass modifizierende Methodenaufrufe möglich sind. Bei Sybase ist dies nicht möglich, da ein `select`-Befehl vor dem Aufruf einer Methode von jedem selektierten Objekt eine Kopie erzeugt. Die aufgerufene Methode ändert dann allenfalls die Kopie (welche an den Client ausgeliefert wird) jedoch nicht das Objekt in der Tabelle, woher es ursprünglich stammt.

Der Dereferenzierungsoperator für Felder eines Objektes ist `'.'`. Da in speziellen Fällen Mehrdeutigkeiten mit dem Besitzernamen einer Tabelle entstehen können, bietet Sybase beispielsweise auch `'>>'` als Dereferenzierungsoperator an.

Unterklassen

- Bezeichnet ein Tabellenattribut eine Java-Klasse, können darin auch alle Unterklassen dieser Klasse gespeichert werden. Beispiel

```
public class PersonenAdresse extends Adresse
{
    public String zuhanden;
    public PersonenAdresse( String strasse, String ort,
                           String plz, String zuhanden) {};
}
```

- Beispieldaten

```
insert into Person ( adresse )
  values ( new Adresse('Auweg 3', '3253', 'Neuenegg' ) )
insert into Person ( adresse )
  values ( new PersonenAdresse('Eystre 3', '3006',
                                'Bern', 'Moser' ) )
```

select mit Unterklassen

- Eine Abfrage bezieht sich immer auf den *deklarierten* Typ. Folgende Abfrage liefert einen *Fehler*:

```
select adresse.zuhanden from Person
```

Das Attribut `zuhanden` ist in der Klasse `Adresse` nicht vorhanden.

- *Korrekt* ist daher folgende Abfrage:

```
select (cast(adresse as PersonenAdresse)).zuhanden
from Person
where adresse.getClass().getName() = 'PersonenAdresse'
```

Die `cast` Funktion führt hier eine Konversion von einer Ober- in eine Unterklasse durch. Da diese Operation vom Typsystem her nicht zwingend korrekt ist, muss mit einer `where`-Bedingung sichergestellt werden, dass nur konvertierbare Objekte aus der Datenbank gelesen werden.

Die etwas merkwürdige Syntax der `cast`-Funktion rührt daher, dass mit ihr auch gewöhnliche Wert-Konversionen innerhalb von SQL vorgenommen werden können, beispielsweise `cast( idPerson as varchar(20) )`.

Ändern mit update

- Die portabelste Variante ersetzt nur *ganze* Objekte innerhalb eines update-Befehls:

```
update Person set adresse = new Adresse(...)  
where idPerson = 1
```


resp in JDBC:

```
update Person set adresse = ? where idPerson = 1
```
- Die JDBC-Variante ist sicher die häufigste: Das Objekt wird mit `select` *ganz* gelesen, in der *Applikation* modifiziert und mit `update` *ganz* wieder zurückgeschrieben.
- Gemäss SQL-J ist das *direkte* Ändern von Objekt-Attributen möglich:

```
update Person  
set adresse.strasse = 'neue strasse'  
where idPerson = 1
```

Gewisse Produkte (Cloudscape) lassen das Ändern einzelner Attribute mit dem update-Befehl nicht zu. In diesem Fall muss entweder ein neues Objekt mit den neuen Datenwerten erzeugt werden, oder das Objekt in die Applikation gelesen, dort verändert und anschliessend zurückgeschrieben werden.

Ändern mit update

- Attribute in Objekten werden meist über Methoden geändert. Eine entsprechende Set()-Methode hat beispielsweise die Form:

```
void setOrt( String ort )
```

- Funktioniert folgender Aufruf?

```
update Person
set adresse = adresse.setOrt( 'neuer Ort' )
where idPerson = 1
```

- Der Aufruf funktioniert nur, wenn die Datenbank beim Laden der Klasse Methoden mit `void` Rückgabe in Methoden mit `this` Rückgabe ändert.

Wenn das Datenbanksystem eine Klasse beim Laden unverändert übernimmt (beispielsweise Cloudscape), funktioniert folgender Befehl *nicht*, weil die Methode ja nicht ein Objekt zurückliefert, das dem Feld `msg` zugewiesen werden kann:

```
update Person
set adresse = adresse.setOrt( 'neuer Ort' )
where idPerson = 1
```

Es muss eine *neue* Modifikationsmethode entworfen werden, welche `this` zurückgibt, beispielsweise:

```
public class Adresse {
    ...
    public Adresse updateOrt( String ort ) {...; return this; }
}
```

Damit wäre folgender SQL-Befehl möglich:

```
update Person
set adresse = adresse.updateOrt( 'neuer Ort' )
where idPerson = 1
```

Löschen mit delete

- Das Löschen von Datensätzen ist von SQL-J wenig betroffen, ausser bei der Formulierung der where-Bedingung.
- Beispiel:

```
delete from Person  
where adresse.strasse like = 'Auweg%'
```

null - Werte

- Der Umgang mit null-Werten, wie man sie von SQL her kennt, funktioniert analog in SQL-J

- Beispiel für Tabellendefinition:

```
create table Person (  
    idPerson integer default autoicrement,  
    adresse Adresse not null,  
    ...  
)
```

- Beispiel einer Bedingung mit Test auf null:

```
where adresse is not null
```

Tabellenattribute dürfen als nullable definiert sein (with null als Zusatzangabe nach dem Datentyp). Bei einem update- oder insert-Befehl darf in diesem Fall Wert null eingefügt werden.

In der where-Klausel eines SQL-Befehles darf die Bedingung is null oder is not null vorkommen. Sie gilt in gleicher Weise für SQL-Typen wie für Java-Objekte.

Statische Methoden und Attribute

- Auf Klassenmethoden und Klassenattribute kann via SQL zugegriffen werden. Beispiele:

```
select Adresse.getOrtByPlz( '3006' )
```

```
select Adresse.getOrtByPlz( adresse.getPlz() )  
from Person
```

```
select Adresse.plzFormat
```

- Klassenattribute können geändert werden, unterliegen aber in Sybase keiner Transaktionssemantik. Der aktuelle Wert geht bei Terminierung des DB-Servers verloren.
- Der SQL-J Standard lässt die Transaktionssemantik von Klassenattributen offen.
- Klassenmethoden können ein Ersatz für Stored Procedures sein.

`getOrtByPlz()` sei eine statische Klassenmethode und `plzFormat` sei ein statisches Klassenattribut.

Der offizielle Operator für die Referenzierung einer Klassenmethode ist `::`. Meistens wurde jedoch die Java-Syntax mit `.` übernommen.

Transiente Attribute

- Transiente Attribute gehören zur Klassendeklaration und können abgefragt werden, sind aber *immer* mit dem von Java vorgegebenen Defaultwert belegt.
- Neue Werte für transiente Attributen werden ohne Fehler entgegengenommen, bleiben aber *wirkungslos*.
- Beachte: statische Klassenattribute können im Server für Anwendungszwecke zeitweilig geändert werden, transiente Objekt-Attribute nie!

Aufgaben

- Erstellen Sie eine Klasse `SQLJObject` mit folgenden Elementen:
 - `String`, `Datum` und `Vector` **Attribute**
 - `public` und `private` **Attribute**
 - `static` und `transient` **Attribute**
 - lesende und modifizierende Zugriffsfunktionen
 - `toString()` **Methode**
 - eine Funktion, welche eine `Exception` wirft
 - eine abgeleitete Klasse `MySQLJObject`.
- Erstellen Sie eine Tabelle mit einem Feld vom Typ `SQLJObject` und fügen sie ein paar Datensätze ein mit Objekten der Basisklasse und der abgeleiteten Klasse `MySQLJObject`.
- Testen sie `select`-und `update`-Befehle auf die verschiedenen Elemente.
- Wie greifen Sie auf Objekte der abgeleiteten Klasse zu?

Ordnung auf Datentypen

- Viele Abfragen basieren auf Datentypen, die eine Ordnung voraussetzen. Beispiele:

```
select * from Person where gebdatum > '2002-01-31'  
select * from Person order by name  
select min(gebdatum) from Person
```
- Die meisten SQL-Datentypen besitzen eine natürliche, applikations-unabhängige Ordnung. Folgende Operatoren und Befehle können beispielsweise auf Zahlen, Datumswerte und Strings angewendet werden:

```
>, <, >=, <=  
min(), max(), order by  
create index
```

Klassen haben keine Ordnung

- Klassen besitzen meist keine implizite Ordnung. Entsprechende Operatoren und Befehle können deshalb nicht ohne Weiteres angewendet werden:

```
select adresse from Person
order by adresse ←———— ?
```

- Attribute und Methoden, welche einen Datentypen mit einer Ordnung haben, können jedoch für Vergleiche und Sortierungen verwendet werden:

```
select adresse from Person
where adresse.getPlz() >= '3000'
select adresse from Person
order by adresse.toString()
```

An sich kann jede Sortierung und jeder Vergleich von Objekten durch Benutzung einer Methode oder eines Attributes realisiert werden, welches einen Basistyp zurückliefert. Der Basistyp muss lediglich selbst eine Ordnung definieren, was für SQL- und Java-Basistypen fast immer der Fall ist. Java-Klassen, die im Wesentlichen Basistypen darstellen (Integer, Date, String etc.) haben ebenfalls eine Ordnung (siehe nächste Seite).

Ein ganz wesentliches Problem ist die Performance obiger where-Klausel. Sobald ein Objekt für die Vergleichsoperation ausgepackt werden muss, kann nicht mehr via Index gesucht werden. Es ist auch nicht möglich, auf Attribute, resp. Methoden eines Objektes einen Index zu erstellen. Die Folge ist, dass für das Auffinden der Objekte, welche der Bedingung genügen, die ganze Tabelle durchgearbeitet werden muss. Der Suchaufwand ist proportional zur Tabellengrösse. Bei Bedingungen, welche mehrere Tabellen überspannen, kann der Suchaufwand noch grösser sein. Dasselbe gilt für die Sortierung der Ausgabe.

Klassen eine Ordnung geben

- Eine Ordnung auf einer Klasse muss *spezifisch* für diese Klasse *definiert* werden.
- In Java wird eine Ordnung definiert durch Implementation des Comparable-Interface:

```
Interface java.lang.Comparable
{
    int compareTo( Object o ) throws ClassCastException;
}
```

- Für einfache Java Standard-Klassen (String, Date, Integer, Float, Byte, Char etc.) ist Comparable **bereits implementiert**.

Eine Ordnung braucht nur definiert zu werden, wenn Befehle und Operatoren für geordnete Datentypen verwendet werden sollen. Rein für die Speicherung von Objekten wird keine Ordnung benötigt.

`compareTo()` liefert -1, 0, oder +1 zurück, je nachdem was der Vergleich des eigenen Objektes mit dem Parameter-Objekt `o` ergibt. `compareTo()` kann vom Entwickler nach Belieben programmiert werden. Weil für jede Klasse, die Comparable implementiert entscheidbar ist, ob ein Objekt grösser, gleich gross oder kleiner als ein anderes derselben Klasse ist, kann auch ein Index erstellt werden. Where-Bedingungen mit Vergleichsoperatoren, order by-Klauseln sowie `min()` und `max()`-Funktionen können nun optimiert abgearbeitet werden.

Eine der Hauptschwierigkeiten ist, dass für jede Java-Klasse nur *eine* Ordnung für den Gebrauch in der Datenbank festgelegt werden kann. Für die Klasse Adresse gäbe es eventuell mehrere denkbare Ordnungen, die alle in einer Applikation gebraucht werden.

Anwendungsbeispiel für Comparable

```
public class Adresse implements Serializable, Comparable
{
    ...
    public int compareTo( Object o ) throws ClassCastException
    {
        Adresse a = (Adresse) o;
        String s1 = this.getPlz() + this.strasse;
        String s2 = a.getPlz() + a.strasse;
        return s1.compareTo( s2 );
    }
}
```

Damit wird beispielsweise die SQL-Abfrage möglich:

```
select adresse from Person
order by adresse
```

`compareTo()` für die Adress-Klasse vergleicht den gesamten Adress-Text auf Gleichheit. Beachte, dass für die Klasse `String` die Methode `compareTo()` bereits implementiert ist und genutzt werden kann. Praktischerweise könnte man in obigem Beispiel noch die Funktion `toUpperCase()` benutzen, um das Vergleichsergebnis unabhängig von der Gross-/Kleinschreibung zu machen.

Um die Signatur der Methode `compareTo()` immer gleich zu halten, wird ein Objekt der Klasse `Object` übergeben. Im konkreten Fall ist `Object` natürlich meistens ein Objekt der eigenen Klassen. Der cast-Operator `(Adresse)` stellt dies sicher, ansonsten wird eine `ClassCastException` ausgeworfen.

Sobald eine Klasse das Interface `Comparable` implementiert hat, werden auch Vergleiche mit `=` und `!=` darüber abgewickelt.

Dringend empfohlen ist, die Methoden `equals()` und `hashCode()` sinngemäss zu `compareTo()` zu implementieren. Sie werden unter Umständen vom DBMS für die Vergleichsoperationen `'='` und `'<>'` benutzt.

```
public boolean equals( Object o ) throws ClassCastException
{
    Adresse a = (Adresse) o;
    return ( this.getPlz().equals( a.getPlz() )
            && this.strasse.equals(a.strasse ) );
}

public int hashCode()
{
    return (this.getPlz() + this.strasse ).hashCode();
}
```

Mehr Funktionalität mit Comparable

Comparable ermöglicht es, Vergleichsoperatoren mit mehr "Wissen" über die Semantik der Klasse zu versehen. Beispiel:

```
public class Reservation implements Serializable,
    Comparable
{
    protected String was;
    protected Date von;
    protected Date bis;
    public Reservation( String was, String von, String
        bis){};
    public int compareTo( Object o ) {}
}
```

Die Möglichkeiten sind vergleichbar mit dem Überladen von Operatoren in C++.

In obigem Beispiel ist eine Reservation dann *gleich* wie eine andere, wenn sich die Reservations-Perioden überschneiden. Eine Reservation ist *kleiner* als eine andere, wenn sie *ganz vor* der anderen Reservation liegt. Die Implementation dieses Vergleichs ist einfach und ermöglicht beispielsweise folgende Abfragen in einer Reservations-Tabelle:

```
select reservation from Reservation
where reservation =
    new Reservation('Zi 313', '2002-12-18', '2003-01-6');
```

Mit dieser Abfrage wird festgestellt, ob sich die neue Reservation mit einer bestehenden überschneidet.

Weitere Beispiele:

- Eigene Stringklasse, die beim Vergleich Gross-/Kleinschreibung unterdrückt.

Exceptions

- Exceptions von Java-Methoden, welche innerhalb eines SQL-J Befehles auftreten, werden von der Datenbank an die Applikation weitergegeben.
 - Allgemeine Java Exception werden in eine SQLException mit SQLState = 38000 verpackt.
 - "For portability, a java method executed in an SQL system should re-throw any SQL exception that it catches".
- Benutzerdefinierte SQLExceptions dürfen für SQLState die Werte 38NNN verwenden.
- Für SQL-J sind einige spezielle SQLSTATE-Werte reserviert.

Speziell für SQL-J definierte SQLSTATE Fehlercodes gemäss SQL-J Standard:

Java DDL 46 Invalid URL 001
Java DDL 46 Invalid jar name 002
Java DDL 46 Invalid class deletion 003
Java DDL 46 Invalid jar name 004
Java DDL 46 Invalid replacement 005
Java DDL 46 Invalid grantee 006
Java DDL 46 Invalid signature 007
Java DDL 46 Invalid method specification 008
Java DDL 46 Invalid REVOKE 009
Java execution 46 Invalid null value 101
Java execution 46 Invalid jar name in path 102
Java execution 46 Unresolved class name 103
Java execution 46 Too many result sets 104
Uncaught Java exception 38 (no subclass) 000
User-defined 38 User-defined mmm

JDBC in der Datenbank

- In SQL aufgerufene Java-Methoden können selbst JDBC-Aufrufe enthalten! Die JDBC-Aufrufe werden also *serverseitig* durchgeführt.
- Anwendung 1
 - Eine Java-Methode zur Erzeugung von Schlüsselwerten, die in insert-Befehlen verwendet werden kann.
 - läuft serverseitig im Kontext einer umgebenden Transaktion ab.
- Anwendung 2
 - Periodisch laufender Task (Thread) in der Datenbank.
 - läuft in einer eigenen Transaktion.
- Anwendung 3
 - Eine Stored Procedure in Java geschrieben, welche eines oder mehrere ResultSets an den aufrufenden Client zurückliefert.

Anwendung 1 ist eine typische Aufgabe, welche klassischerweise mit Stored Procedures realisiert wurde. Ein vollständiges Beispiel ist zu finden unter `KeyStoreManager.java`.

Anwendung 2 ist neuartig, indem der Datenbank Aufgaben übertragen werden, die vollständig unabhängig von einem umgebenden SQL-Befehl sind und selbstständig ablaufen. Ein vollständiges Beispiel ist zu finden unter `DatabaseThread.java`.

Anwendung 2 wird von Sybase insofern eingeschränkt, dass keine unabhängigen Threads im Datenbank-Server existieren können. Der SQL-Befehl, welcher den Thread startet, muss also –unendlich– laufend bleiben. In Cloudscape ist Anwendung 2 problemlos möglich.

Anwendung 3 ersetzt Stored Procedures, welche in einer proprietären Programmiersprache (PLSQL bei Oracle, Transact-SQL bei Sybase usw.) geschrieben sind. Anwendung 3 ermöglicht es auch, Java nur innerhalb der Datenbank zu benutzen, nach aussen jedoch gewöhnliche SQL-Datentypen auszuliefern. Ein vollständiges Beispiel ist zu finden unter `Procedures.java`.

Demos

- BenchmarkingProducer
 - Overall-Performance für das Erzeugen von Messages
 - Späterer Vergleich mit OODB
- KeyStoreManager
 - Verwendung als Schlüsselgenerator
 - Performance-Vergleich via benchmark() server-sided und main() client-sided
- DatabaseThread
 - Client-unabhängige Aufgaben im Server
- Stored Procedures
 - Rückgabe eines ResultSet verwenden.

Zusammenfassung SQL-J

- + Gleiche Sprache für Applikation und Datenhaltung.
- + Java ist objektorientiert, SQL nicht:
 - Objekte speichern, statt atomare Werte
 - get()- und set()- Methoden anwendbar, statt Direktzugriff
 - Kapselung, Vererbung und Polymorphismus nutzbar
- + Kompakterer Tabellenbestand, weil bessere Strukturierung.
- + Kombination zweier mächtiger Werkzeuge: SQL und Java.

Zusammenfassung SQL-J ff

- Datenbank ist stark an Java Clients gebunden.
- Vorsicht: zuviele Java-Objekte verderben den Brei!
- Optimierbarkeit von Abfragen über Java-Typen heikel. Es kann eine schlechte Performance resultieren.
- Objekte sind mit SQL update-Befehl nur als Ganzes ersetzbar. Problem bei grossen Objekten.
- GUI/Applikationsbuilder können nicht mit Java-Objekten umgehen(?)

Java-Objekte in der DB ohne SQL-J

- Beliebige Java Objekte können in der Datenbank abgelegt werden, wenn sie serialisierbar sind:

Serialisierung/Deserialisierung in Byte-Arrays durch die Applikation. Verwendung des Datentyps LONG VARBINARY für das zugehörige Feld in der Datenbank.

Java-Objekte in der DB ohne SQL-J

- Schreiben in die Datenbank

```
...
String sqlstr = "insert into myTable(binaryfield)
                values ( ? )";
PreparedStatement pstmt = con.prepareStatement ( sqlstr );
ByteArrayOutputStream bo = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream( bo );
out.writeObject( objectToSerialize );
out.close();
pstmt.setBytes( 1, bo.toByteArray() );
pstmt.executeUpdate();
...
```

Java-Objekte in der DB ohne SQL-J

- Lesen aus der Datenbank

```
...
String sqlstr = "select binaryfield from myTable";
PreparedStatement pstmt = con.prepareStatement ( sqlstr );
ResultSet rs = pstmt.executeQuery();
while ( rs.next() ) {
    Byte[] b = rs.getBytes( 1 );
    ByteArrayInputStream bi = new ByteArrayInputStream( b );
    ObjectInputStream in = new ObjectInputStream( bi );
    objectToDeserialize = (MyClass) in.readObject();
}
...
```