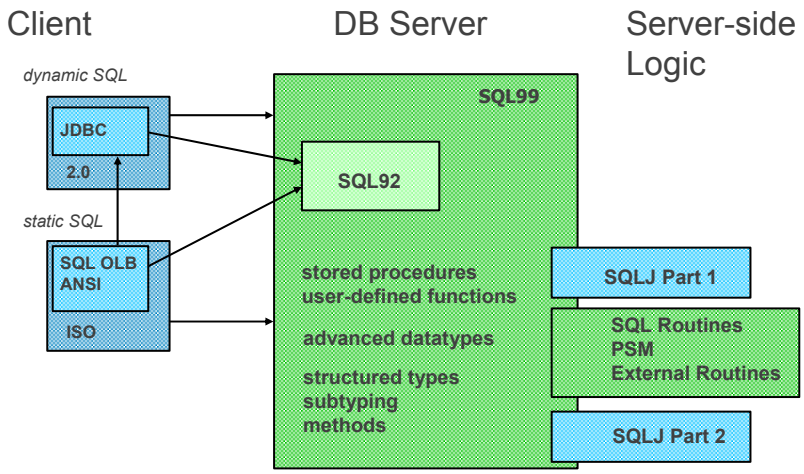
Chapter 5 – Object-Relational SQL and Java

Prof. Dr.-Ing. Stefan Deßloch
AG Heterogene Informationssysteme
Geb. 36, Raum 329
Tel. 0631/205 3275
dessloch@informatik.uni-kl.de

Inhalt

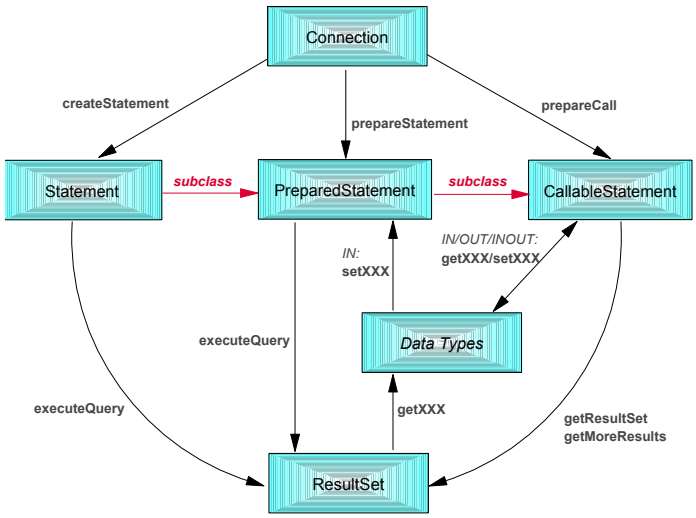
The "Big Picture"



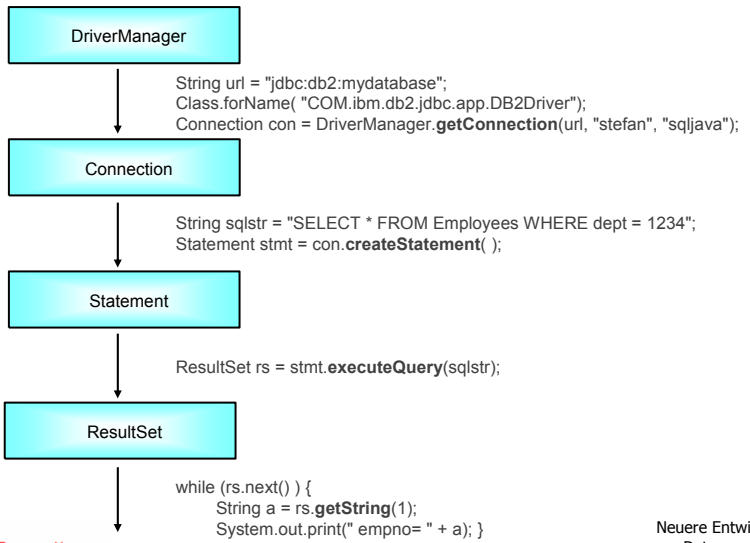
JDBC

- De-facto standard for Java data access
  - specification owned by Sun
  - Java core API (JDK 1.1 java.sql package)
  - major SQL database vendor support
- Call-level, dynamic SQL API
- Concepts derived from ODBC/CLI
  - can be implemented on top of ODBC/CLI layer
- SQL Conformance
  - requires SQL92 Entry Level
  - permits extended SQL, vendor-specific dialects

JDBC – Major APIs



JDBC API

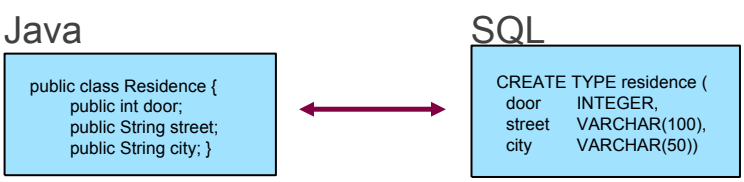


Extensions for Object-Relational Types

- Added support for object-relational data types
  - User-defined types (UDTs)
    - Structured types
    - Distinct types
  - References
  - Arrays (collection types)
- Supports mappings of DB user-defined types to/from Java classes
  - Focus on object state, not on interface (behavior)
  - Provides sufficient basis for mapping tools
  - Allows application to provide mapping information to JDBC driver

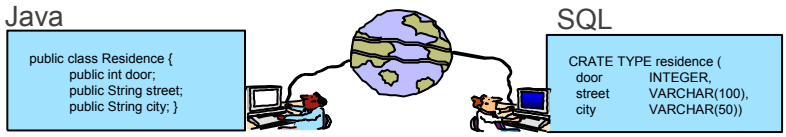
JDBC 2.0 Structured Type Support

- Materializing SQL99 types as Java objects
  - SQL99 types manipulated using existing result set or prepared statement interfaces
  - get/setObject(<column>) simply "works" for structured types
  - Example:
    ResultSet rs = stmt.executeQuery("SELECT e.addr FROM Employee e");
    rs.next( );
    Residence addr = (Residence)rs.getObject(1);



Mapping Infrastructure

- Mapping table for recording correspondence of DB UDT and Java class
  - JDBC driver automatically generates client object, invokes method to 'internalize' state.
  - Can be attached to a DB connection object
  - Can be used as additional parameter in get/setObject() calls
- Java class implements interface SQLData
  - readSQL() reads attributes from an SQLInput data stream
  - writeSQL() writes attributes to an SQLOutput data stream
  - Ordering of attributes has to be preserved during read/write
  - Includes handling of nested objects, type conversions, NULL attributes
- SQLInput, SQLOutput interfaces
  - Generic 'stream-based' API for implementing the customized mapping
  - Used by programmers and mapping tools
  - Vendor-specific implementation details of object bind-out are hidden



Mapping (Example)

- **Java class**

```
public class Residence implements SQLData {
    public int door;
    public String street;
    public String city;
    public void readSQL(SQLInput stream, ...) throws SQLException {
            door = stream.readInt();
            street = stream.readString();
            city = stream.readString(); }
    public void writeSQL(SQLOutput stream, ...) throws SQLException {
            stream.writeInt(door);
            stream.writeString(street);
            stream.writeString(city); } }
```

- **SQL99 type**

```
CREATE TYPE residence (
    door    INTEGER,
    street  VARCHAR(100),
    city    VARCHAR(50))
```

- **JDBC driver (SQLJ) automatically generates client object**
  invokes method to 'internalize' state.Java class
- **Mapping table**
  records correspondence DB2 type/Java class
- **Server-side SQL99 transformation**
  defines how UDT is passed to/from the client

Structured Types: Default Mapping

- Uses new JDBC interface 'Struct'

```
Struct st = (Struct)resultset.getObject(1)
public interface Struct extends SQLData {
  SQLType getSQLType();
  Object[ ] getAttributes();
}
```

  - ResultSet.getObject() will now return an object implementing the Struct interface
- JDBC driver includes a new Java class implementing the Struct interface
- Generic way of handling a structured object as an array of Java objects that represent the individual attribute values
  - Useful for generic applications/tools

Object References

- New methods on ResultSet, PreparedStatement

```
Ref ref = rs.getRef(1);
```

- Ref interface
  - Has method for determining the (static) type of the referenced object
    - Hides the underlying data type of the reference
- A Ref object can be used as a parameter in other SQL statements
  - Dereference
  - Path expressions
  - Updates
  - ...

Arrays

- Retrieving/storing arrays
  - get/setArray() methods on ResultSet, PreparedStatement
  - Array interface supports methods to:
    - Determine the element type
      - Retrieve an array as a Java array, list of Java objects
    - Open a result set on an array (i.e., turn array into a table)
  - Implementation based on array locators

SQL Object Language Bindings (OLB)

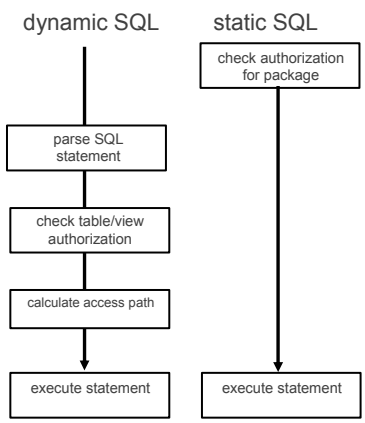
- aka SQLJ Part 0
- Static, embedded SQL in Java
  - Development advantages over JDBC
    - more concise, easier to code
    - static type checking, error checking at precompilation time
  - SQLJ translator implemented using JDBC
  - Permits static authorization
  - Promises performance benefits through vendor-specific customizer
- Accepted ANSI/ISO standard as part of SQL
  - "Database Language - SQL, Part 10 Object Language Bindings (SQL/OLB)", ISO/IEC 9075-10:2000
  - Based on SQL-99, JDBC 2.0

SQL OLB Overview

- Static SQL syntax for Java
  - INSERT, UPDATE, DELETE, CREATE, GRANT, etc.
  - Singleton SELECT and cursor-based SELECT
  - Calls to stored procedures (including result sets)
  - COMMIT, ROLLBACK
  - Methods for CONNECT, DISCONNECT
- SQLJ programs are smaller than JDBC applications
- Can be used in client code and stored procedures
  - Easier than JDBC
- Binary portability

SQL OLB - Static SQL

- Static SQL authorization option
  - Static SQL is associated with "program"
    - Plans/packages identify "programs" to DB
    - Program author's table privileges are used
    - Users are granted EXECUTE on program
  - Dynamic SQL is associated with "user"
    - No notion of "program"
    - End users must have table privileges
    - BIG PROBLEM FOR A LARGE ENTERPRISE !!!
- Tradeoffs
  - Static vs. dynamic
  - SQLJ vs. JDBC
  - Less flexible at run-time
  - Allows error checking at development time
  - Static SQL can be faster!!!



SQLJ vs. JDBC: Retrieve Single Row

- SQLJ

```
#sql [con] { SELECT ADDRESS INTO :addr FROM EMP
             WHERE NAME=:name };
```

- JDBC

```
java.sql.PreparedStatement ps = con.prepareStatement(
             "SELECT ADDRESS FROM EMP WHERE NAME=?");
ps.setString(1, name);
java.sql.ResultSet names = ps.executeQuery();
names.next();
name = names.getString(1);
names.close();
```


Result Set Iterators

- Mechanism for accessing the rows returned by a query
  - Comparable to an SQL cursor
- SQLJ Iterator declaration clause results in generated iterator class
  - Iterator is a Java object
  - Iterators are strongly typed
  - Generic methods for advancing to next row
- SQLJ assignment clause assigns query result to iterator
- Two types of iterators
  - Named iterator
  - Positioned iterator

User-defined Types - Example

- assume distinct type ZIPCODE, structured type ADDRESS with subtypes HOME and BUSINESS
- file addrpckg/addressmap.properties:

```
# file: addressmap.properties
class.addrpckg.Address = STRUCT ADDRESS
class.addrpckg.BusinessAddress = STRUCT BUSINESS
class.addrpckg.HomeAddress = STRUCT HOME
class.addrpckg.ZipCode = DISTINCT ZIPCODE
```

- context declaration refers to addressmap:

```
#sql context Ctx with (typeMap = "addrpckg.addressmap");
```

- assume the following table exists:

```
CREATE TABLE PEOPLE (
                FULLNAME CHARACTER VARYING(50),
                BIRTHYEAR NUMERIC(4,0),
                ADDR ADDRESS )
```

- iterator declaration for PEOPLE uses Java Address type:

```
#sql public iterator ByPos (String, int, addrpckg.Address);
```


User-defined Types - Example (cont.)

- sample program for retrieving Address objects:

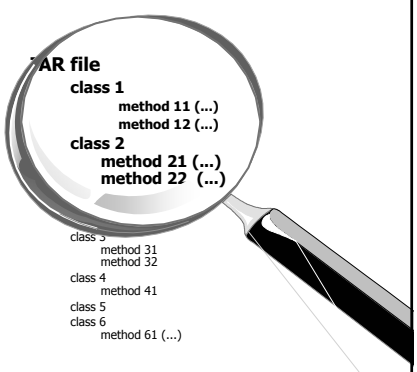
```
{
  ByPos positer; // declare iterator object
  String name = null;
  int year = 0;
  addrpckg.Address addr = null;
  String url;
        ...
  Ctx context = new Ctx(url, false);
  // populate it
  #sql [context] positer = { SELECT FULLNAME, BIRTHYEAR, ADDR FROM PEOPLE };
  #sql { FETCH :positer INTO :name, :year, :addr};
  while ( !positer.endFetch() )
  {
       System.out.println ( name + " was born in "
                  + year + " and lives in " addr.print() );
       #sql { FETCH :positer INTO :name, :year, :addr};
  }
}
```

SQLJ Part 1

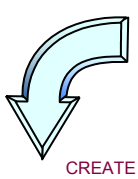
- SQL Routines using the Java™ Programming Language
  - Java static methods used to implement SQL stored procedures and user-defined functions
    - parameter type conversion, error/exception handling
    - stored procedures: output parameters, returning result sets
    - body can contain JDBC, SQLJ
  - SQL DDL statement changes
    - create procedure, create function
  - JAR file becomes a database "object"
    - built-in procedures to install, replace, remove JAR file in DB
    - usage privilege on JAR files
- Accepted ANSI standard
  - ANSI NCITS 331.1:1999
- Has been folded into SQL:2003 Foundation

Installing Java Classes in the DB

- Installation
  - New install_jar procedure sqlj.install_jar ('file:~/classes/routines.jar', 'routines_jar')
  - Parameters: URL of JAR file with Java class and string to identify the JAR in SQL
  - Install all classes in the JAR file
  - Uses Java reflection to determine names, methods, signatures
  - Optionally uses deployment descriptor file found in JAR to create SQL routines
- Removal
  - sqlj.remove_jar ('routines_jar')
- Replacement
  - sqlj.replace_jar ('file:~/classes/routines.jar', 'routines_jar')

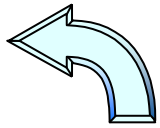


Creating Procedures and UDFs

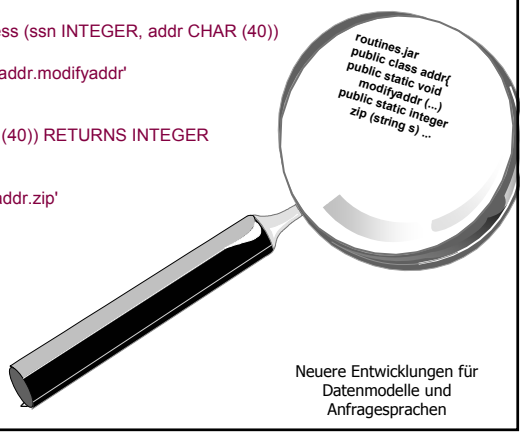


sqlj.install_jar ('file:~/classes/routines.jar', 'routines_jar')

Java return type 'void' -> stored procedure
otherwise -> user-defined function



```
CREATE PROCEDURE modify_address (ssn INTEGER, addr CHAR (40))
    MODIFIES SQL DATA
    EXTERNAL NAME  'routines_jar:addr.modifyaddr'
    LANGUAGE JAVA
    PARAMETER STYLE  JAVA
CREATE FUNCTION zip (addr CHAR (40)) RETURNS INTEGER
    NO SQL
    DETERMINISTIC
    EXTERNAL NAME 'routines_jar:addr.zip'
    LANGUAGE JAVA
    PARAMETER STYLE  JAVA
```



SQLJ Stored Procedures

- OUT and INOUT parameters
  - CREATE PROCEDURE avgSal (IN dept VARCHAR(30), OUT avg DECIMAL(10, 2)) ...
  - Java method declares them as arrays
  - Array acts as container that can be filled/replaced by the method implementation to return a value
    - public static void averageSalary (String dept, BigDecimal[ ] avg) ...
- Returning result set(s)
  - CREATE PROCEDURE ranked_emps (region INTEGER) DYNAMIC RESULT SETS 1 ....
  - Java method declares explicit parameters for returned result sets of type
    - array of (JDBC) ResultSet
    - array of (SQLJ) iterator class, prev. declared in "#sql iterator ..."
    - public static void ranked_emps (int region, ResultSet[ ] rs) ...
  - Java method body assigns (open) result sets as array elements of result set parameters
  - Multiple result sets can be returned

Error Handling

- Java method throws an SQLException to indicate error to the SQL engine
  - ... throws new SQLException ("Invalid input parameter", "38001");
  - SQLSTATE value provided has to be in the "38xxx" range
- Any other uncaught Java exception is turned into a SQLException "Uncaught Java exception" with SQLSTATE "38000" by the SQL engine
- Java exceptions that are caught within an SQLJ routine are internal and do not affect SQL processing

Additional Features

- Java "main" methods
  - Java signature has to have single parameter of type String[]
  - Corresponding SQL routine has
    - Either 0 or more CHAR/VARCHAR parameters,
    - or a single parameter of type array of CHAR/VARCHAR
- NULL value treatment
  - Use Java object types as parameters (see JDBC)
    - SQL NULL turned into Java null
  - Specify SQL routine to return NULL if an input parameter is NULL

    ```
    CREATE FUNCTION foo(integer p) RETURNS INTEGER
      RETURNS NULL ON NULL INPUT
    ```

  - Otherwise run-time exception will be thrown
- Static Java variables
  - Can be read inside SQL routine
  - Should not be modified (result is implementation-defined)
- Overloading
  - SQL rules may be more restrictive
  - Map Java methods with same name to different SQL routine names

SQLJ Part 2

- SQL Types using the Java™ Programming Language
- Use of Java classes to define SQL types
  - Can be mapped to structured types or "native" Java types (blobs)
  - Can be used to define columns in tables
  - Can be used to define SQL99 tables (structured types)
- Mapping of object **state** and **behavior**
  - Java methods become SQL99 methods on SQL type
  - Java methods can be invoked in SQL statements
- Automatic mapping to Java object on fetch and method invocation
  - Java Serialization
  - JDBC 2.0 SQLData interface
- Includes handling of USAGE privilege on SQL type
- Use the procedures introduced in SQLJ Part 1 to install, remove, and replace SQLJ JAR files
- Approved ANSI standard
  - ANSI/NCITS 331.2-2000
- Folded into SQL:2003 Foundation

Mapping Java Classes to SQL

- Described using extended CREATE TYPE syntax
  - DDL statement, or
  - Mapping description in the deployment descriptor
- Supported Mapping

Java	SQL
class	user-defined (structured) type
member variable	attribute
method	method
constructor	constructor method
static method	static method
static variable	static observer method

  - SQL constructor methods
    - Have the same name as the type for which they are defined
    - Are invoked using the NEW operator (just like in Java)
  - SQL does not know static member variables
    - Mapped to a static SQL method that returns the value of the static variable
    - No support for modifying the static variable

Mapping Example

- Java class

```
public class Residence implements Serializable, SQLData {
        public int door;
        public String street;
        public String city;
        public static  String country = "USA";
        public String printAddress( ) { ...};
        public void changeResidence(String adr) { ... // parse and update fields ...}
        // SQLData methods
        public void readSQL(SQLInput in, String type) { ... };
        public void writeSQL(SQLOutput out) { ... };
 }
```

- SQL DDL/descriptor statement

```
CREATE TYPE Address EXTERNAL NAME 'residence_jar:Residence' LANGUAGE JAVA (
        number    INTEGER   EXTERNAL NAME 'door',
        street      VARCHAR(100) EXTERNAL NAME 'street',
        city        VARCHAR(50) EXTERNAL NAME 'city',
        STATIC METHOD country( ) RETURNS CHAR(3)
                    EXTERNAL VARIABLE NAME 'country',
        METHOD print() RETURNS VARCHAR(200) EXTERNAL NAME 'printAddress',
        METHOD changeAddress (varchar(200)) RETURNS  Address
                    SELF AS RESULT EXTERNAL NAME  'changeResidence'
)
```


Usage Examples

- Use type as column type

```
CREATE TABLE employees (
        name                VARCHAR(40),
        addr                Address)
```

- Use type for typed table

```
CREATE TABLE addresses OF Address
        (REF IS id SYSTEM GENERATED)
```

- Insert objects

```
INSERT INTO employees VALUES('John Doe', NEW Address( ))
INSERT INTO addresses VALUES(1357, 'Ocean Blvd.', 'Santa Cruz')
Update object
UPDATE employees
SET addr = addr.changeAddress('1234 Parkway Dr., San Leandro')
WHERE name = 'John Doe'
```

- Select object information

```
SELECT addr.print( )
FROM employees
WHERE addr.city = 'San Leandro'

SELECT id->print()
FROM addresses
WHERE city = 'Santa Cruz'
```

Instance Update Methods

- Java and SQL have different object update models
  - Java model is object-based
    - Object method updates object member variables, usually returns void
  - SQL model is value-based
    - Object method returns a modified copy of the object
    - UPDATE statement is required to make object modifcation permanent
- SQLJ permits mapping without requiring modification of Java methods
  - SELF AS RESULT in deployment descriptor identifies an instance update method
  - Java class

```
public class Residence implements Serializable, SQLData {
  ...
  public void changeResidence(String adr) { ... // parse and update fields ...}
}
```

  - SQL type

```
CREATE TYPE Address  EXTERNAL NAME 'Residence' LANGUAGE JAVA (
  ...
  METHOD changeAddress(varchar(200)) RETURNS Address SELF AS RESULT
                                EXTERNAL NAME 'changeResidence'
)
```

- At runtime, the SQL system
  - Invokes the original Java method (returning void) on (a copy of) the object
  - Is responsible for returning the modified object

Object "Conversion" between SQL and Java

- Serializable vs. SQLData
  - Can be specified in CREATE TYPE statement (optional clause)
    - CREATE TYPE Address ... LANGUAGE JAVA USING **SERIALIZABLE** ...
    - CREATE TYPE Address ... LANGUAGE JAVA USING **SQLDATA** ...
  - default is implementation-defined
    - implementation may only support one of the mechanisms
  - does not impact the application program itself
- USING SERIALIZABLE
  - persistent object state entirely defined by Java serialization
  - SQL attributes have to correspond to Java public fields
    - Java field names have to be listed in the external name clauses
  - Java serialization is used for materializing the object in Java
    - attribute access, method invocation
- USING SQLDATA
  - persistent state is defined by the attributes in the CREATE TYPE statement
  - external attribute names do not have to be specified
  - SQLData interface is used for materializing objects in Java
    - read/writeSQL methods have to read/write attributes in the order defined
    - Java fields might be different then the SQL attributes
- Recommendations for portability
  - have Java class implement both Serializable and SQLData
  - Java class should define the complete persistent state as public fields
  - CREATE TYPE statement should have external names, omit *USING*

Additions to SQL99

- CREATE TYPE statement
  - at the type level
    - LANGUAGE JAVA EXTERNAL NAME ... USING ...
  - at the attribute level
    - EXTERNAL NAME ...
  - at the method level
    - EXTERNAL NAME ...
- CREATE ORDERING statement
  - option RELATIVE WITH COMPARABLE INTERFACE
    - if Java class implements java.lang.Comparable