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

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Chapter 4 – Application Programs and Object-Relational Capabilities

Outline

Overview

I. Object-Relational Database Concepts

1. User-defined Data Types and Typed Tables
2. Object-relational Views and Collection Types
3. User-defined Routines and Object Behavior
4. **Application Programs and Object-relational Capabilities**

II. Online Analytic Processing

5. Data Analysis in SQL
6. Windows and Query Functions in SQL

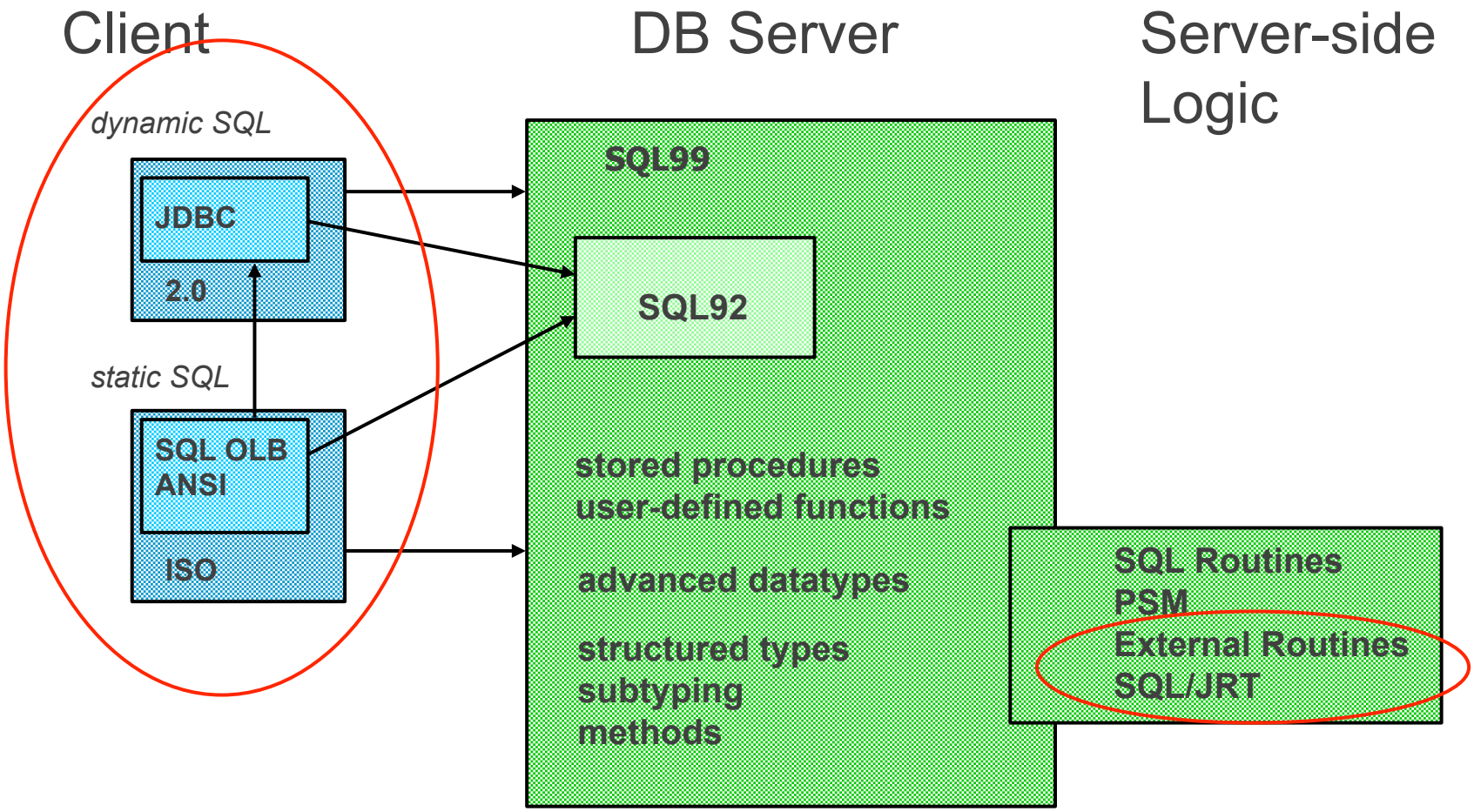
III. XML

7. XML and Databases
8. SQL/XML
9. XQuery

IV. More Developments (if there is time left)

temporal data models, data streams, databases and uncertainty, …

The "Big Picture"



Structured Types and External Programs

- Instance of a structured type has to be made available in an external programming language environment

```
SELECT c.name, c.addr INTO :name, :address
FROM  store s, customers c
WHERE  within(s.loc, :CA)=1 AND
    (within(c.loc, s.zone)=1 OR distance(c.loc, s.loc)<100)
```

client program

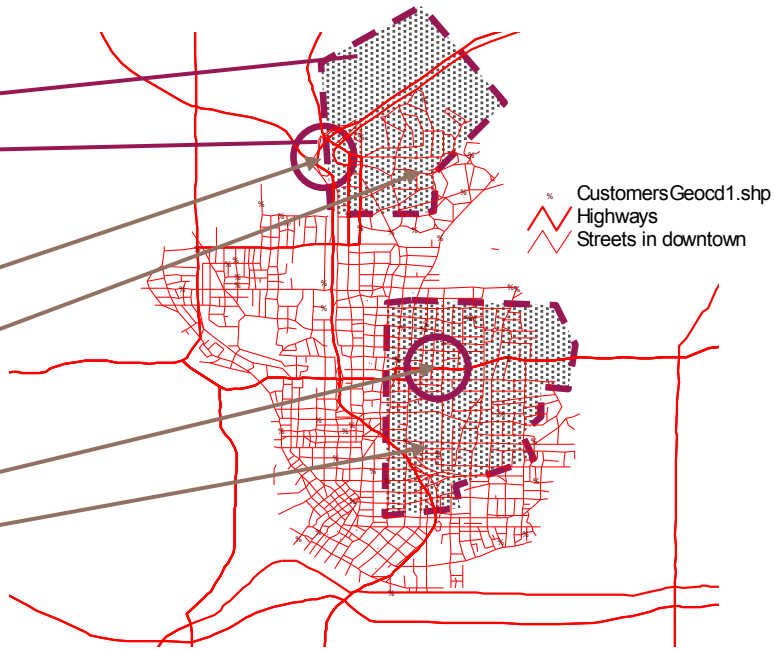
external routine

CUSTOMERS

CID	NAME	INCOME	ADDR	LOC

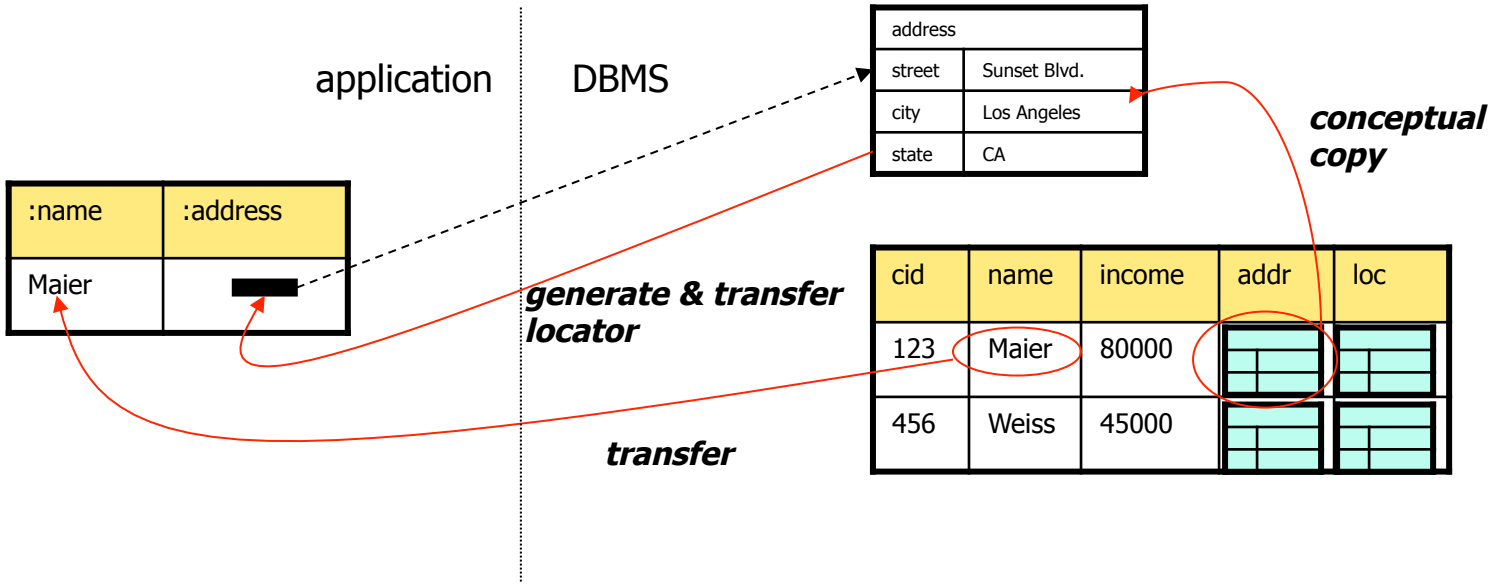
STORES

SID	NAME	ADDR	LOC	ZONE



Approach 1: Locators

- Locator in SQL
  - 'references' an SQL data item that still lives in the SQL engine
    - can be used in SQL statements instead of the data item
    - allow access to the referenced data
  - generated by SQL engine, transferred to application environment
    - application-level concept
    - NOT an SQL data type!
  - first introduced for large objects



UDT, Array and Multiset Locators

- Host variable can be specified as a locator variable for a UDT or an array/ multiset type:

```
SQL TYPE IS point AS LOCATOR pointvar;
SQL TYPE IS INTEGER ARRAY[10] AS LOCATOR avar;
```

- A unique implementation-dependent 4-octet integer locator value is generated and passed to the host variable:

```
EXEC SQL
SELECT center INTO :pointvar
FROM circles WHERE ...
```

- When locators are used in assignment statements, the UDT or the array/ multiset value corresponding to the given locator value is first found, and the result is then used in the assignment:

```
EXEC SQL
UPDATE circles
SET center = :pointvar
WHERE ...
```

Array Support in JDBC

- Based on array locators
- 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)
    - Implicitly executes a
      "SELECT * FROM UNNEST (?)"
      with array locator as parameter

Locators and External Routines

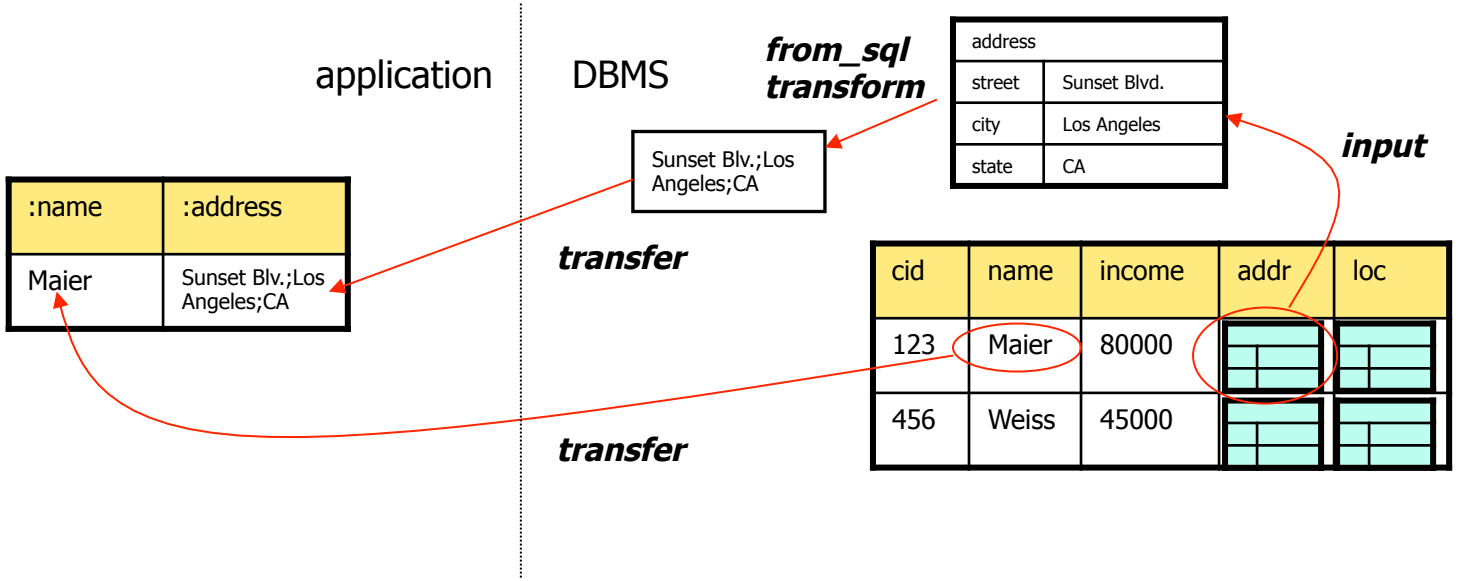
- A parameter of an external routine can be specified as locator parameter if its data type is either a UDT or an array or multiset type, or the returns type of an external function can specify AS LOCATOR if it is either a UDT or an array or multiset type:

```
CREATE FUNCTION foo(p1 emp AS LOCATOR)
RETURNS emp AS LOCATOR
EXTERNAL ...
```

- When the routine is invoked, a unique implementation-dependent 4-octet integer locator value is generated for each input locator parameter and passed as the argument value.
- After the routine finishes execution, for each output locator parameter or function result, the UDT or the array value corresponding to the locator value is first found, and the result is then returned to the caller.

Approach 2: Transforms

- Transforms are user-defined functions or methods that get invoked automatically whenever UDT values are exchanged between SQL and external programs.
- Each UDT is associated with a collection of transform groups; each transform group is associated with:
  - A from_sql function that maps a UDT value into a value of predefined type.
  - A to_sql function that maps a value of a predefined type into a UDT value.



CREATE TRANSFORM

- CREATE TRANSFORM statement specifies a transform for a given UDT

```
CREATE TRANSFORM FOR point
    group1(     FROM SQL WITH FUNCTION from_point1(point),
                TO SQL WITH FUNCTION to_point1(char(27))
    group2(     FROM SQL WITH FUNCTION from_point2(point),
                TO SQL WITH FUNCTION to_point2(char(50));
```

- A transform group with a given name can be specified for only one type within a type hierarchy
- An implicit transform is created for every distinct type on its creation, based on its cast functions
  - User-defined transforms can be created as well

Methods as Transform Functions

- Both from_sql and to_sql functions can be specified as methods:

```
CREATE TRANSFORM FOR point
    group1(     FROM SQL WITH METHOD from_point1() FOR point,
                TO SQL WITH METHOD to_point1(char(27) FOR point)
    group2(     FROM SQL WITH METHOD from_point2() FOR point,
                TO SQL WITH METHOD to_point2(char(50) FOR point);
```

- Both from_sql and to_sql methods can be overridden to define subtype-specific transform methods.
  - dynamic binding rules apply, i.e., if there is an overriding method available, that method is picked for execution.
- If there is no transform available for a UDT with a given group name, then a transform defined for one of its supertypes is picked.

Transforms in Embedded Programs

- An embedded program can specify transform groups for use during the execution of the program:

```
TRANSFORM GROUP group1
TRANSFORM GROUP group2 FOR TYPE point
```

- A host variable whose data type is a UDT must specify a predefined type; must be same as the return type of from_sql function of the transform group specified for the UDT:

```
SQL TYPE IS point AS CHAR(50) pointvar
```

- from_sql function or method is automatically invoked on the UDT value and the result is passed to the host variable:

```
EXEC SQL SELECT center INTO :pointvar FROM circles WHERE ...
```

- to_sql function or method is automatically invoked on the host variable value and the result is passed to SQL:

```
EXEC SQL
   UPDATE circles
   SET center = :pointvar
   WHERE ...
```

Transforms in Dynamic SQL

- SET TRANSFORM GROUP statement sets the transform group for one or more UDTs for use during execution of dynamic SQL statements:

```
SET DEFAULT TRANSFORM GROUP group1;
SET TRANSFORM GROUP FOR TYPE point group2;
```

- Two special registers are provided to inquire about the session defaults:

```
CURRENT_DEFAULT_TRANSFORM_GROUP;
CURRENT_TRANSFORM_GROUP_FOR_TYPE point;
```

Transforms in External Routines

- An external routine can specify transform groups for use during the execution of routine:

```
CREATE FUNCTION foo(p1 point)
RETURNS INTEGER
EXTERNAL
TRANSFORM GROUP group1;
```

- The parameter in the external program corresponding to 'p1' must specify a host language type that corresponds to CHAR(27).
- Transform functions for UDT parameters are picked during the creation of external routines; once selected, the transform functions are frozen.
- Type-preserving functions/methods
  - If a to-sql **method** is defined, then a new instance of the most-specific type of the respective UDT parameter (e.g., SELF) is created, and the to-sql method is invoked on that instance

Dropping Transforms

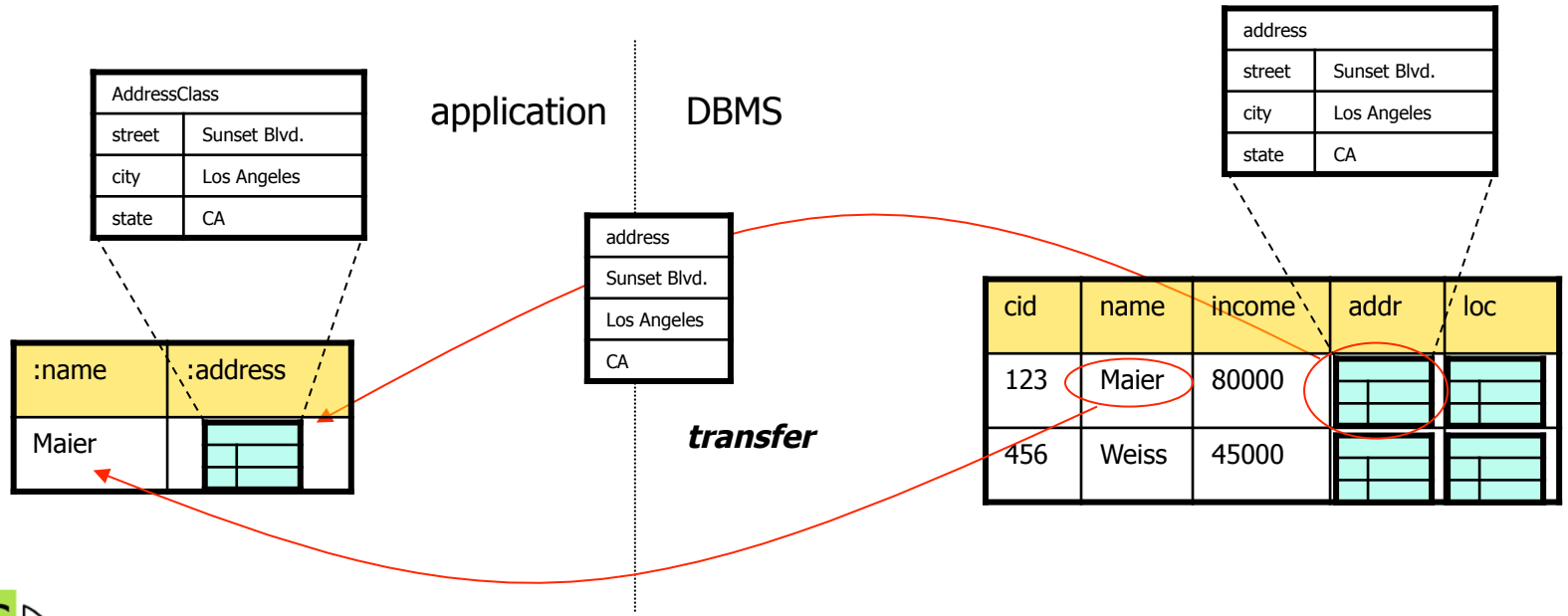
- DROP TRANSFORM statement can be used to drop either a transform group or all transform groups attached to a UDT:

```
DROP TRANSFORM group1 FOR point RESTRICT;
DROP TRANSFORM ALL FOR point CASCADE;
```

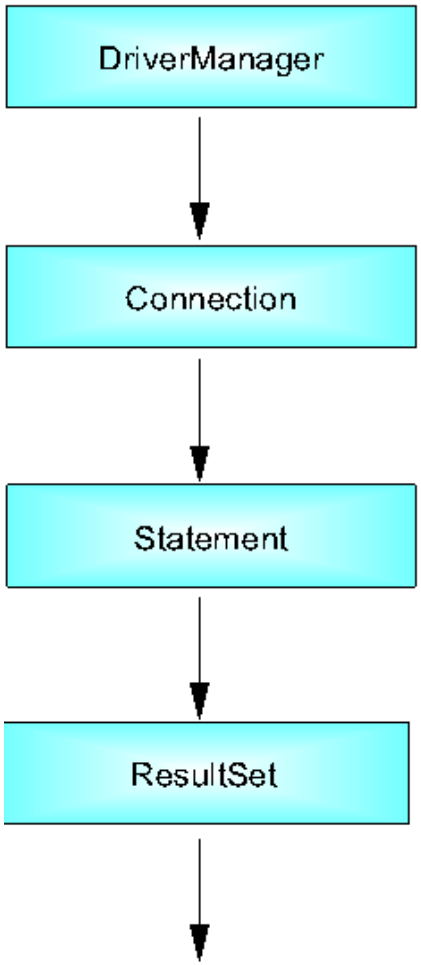
- Dependencies between a transform group and the external routines that depend on that transform group are taken into account during dropping of transforms.

Approach 3: Complex Value Transfer

- Transfer of complex values
  - using proprietary format
  - transparent to application
  - jointly supported by DBMS server and client API (e.g., JDBC driver)
- Generic data/object structures on the application side
  - suitable especially for generic, dynamic applications
- Type-specific mapping for user-defined types



JDBC – Application Program Structure




```
String url = "jdbc:db2:mydatabase";
...
Connection con = DriverManager.getConnection(url, "dessloch",
      "pass");
```

```
String sqlstr = "SELECT * FROM Employees WHERE dept = 1234";
Statement stmt = con.createStatement( );
```

```
ResultSet rs = stmt.executeQuery(sqlstr);
```

```
while (rs.next() ) {
      String a = rs.getString(1);
      String str = rs.getString(2);
      System.out.print(" empno= " + a);
      System.out.print(" firstname= " + str);
      System.out.print("\n");
}
```

Structured Types – Generic Support

- Generic way of handling a structured object as an array of Java objects that represent the individual attribute values
  - Useful for generic applications/tools
- Uses new JDBC interface 'Struct'

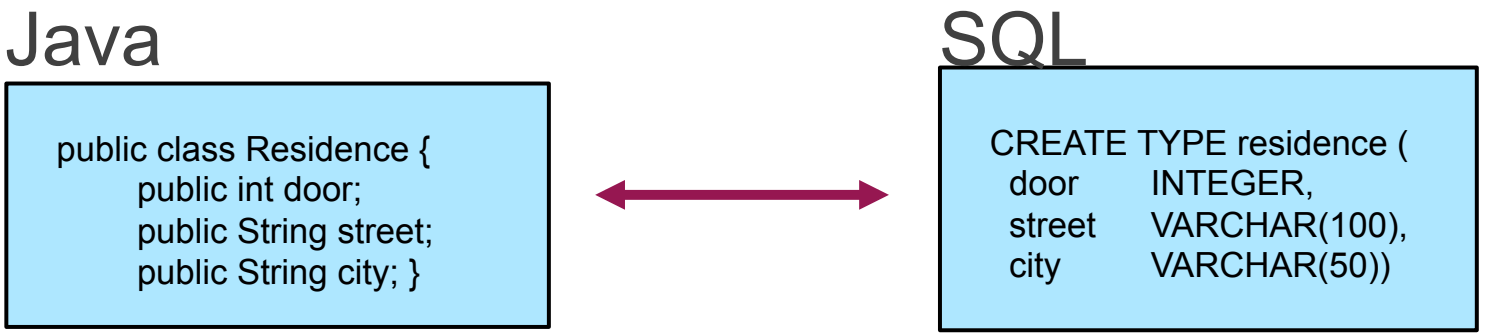
```
public interface Struct extends SQLData {
  String getSQLTypeName();
  Object[ ] getAttributes();
}
```

  - getSQLTypeName() returns the most specific type
  - JDBC driver includes a new Java class implementing the Struct interface
- ResultSet.getObject() will now return an object implementing the Struct interface

```
Struct st = (Struct)resultset.getObject(1)
```

User-defined Type Mapping Support

- Materializing instances of SQL user-defined types as instances of corresponding Java classes
  - 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
  - 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
    - getSQLTypeName() returns corresponding SQL type, used internally by JDBC driver
  - 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
- JDBC driver activities
  - getObject()
    - creates Java object based on type mapping, invokes readSQL(sqlInput) method to 'internalize' state
  - setObject()
    - invokes writeSQL(sqlOutput) to 'externalize' the object state

Mapping (Example)

- SQL99 type

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

- 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); } … }
```

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
- Example:
  - SQL/OLB

```
#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();
```

- Support for composite types, user-defined types based on JDBC
  - in addition, type mapping can be supplied in a properties file

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};
  }
}
```

Retrieving Distinct Types and Using Transforms

- Without a defined mapping
  - distinct and structured types will be transformed into built-in types (transform functions)
  - values are accessed just like for built-in types
- With a mapping defined for distinct or structured types
  - SQLInput/SQLOutput streams will carry only a single value for distinct types
  - same for structured types, if transforms are used

Summary

- Approaches for exchanging complex/collection values with client applications and external routine implementations
  - Locators
    - actual values remain in the SQL environment
    - \+ avoid unnecessary transformation and transfer of complex values
      - performance and development aspect
    - \- restricts value manipulation to SQL operations
    - only approach available for collection types
  - Transform functions for user-defined types
    - \+ high flexibility
      - tailor UDT value exchange to specific application requirements
      - accommodate existing interchange formats
    - \- requires additional development effort
      - transform functions
      - application code for format parsing/generation
  - Complex value transfer for user-defined types
    - \+ generic application representation for dynamic applications
    - \+ user-defined mapping support for improved language integration, productivity
    - (-) potential development impact for application (SQLData) in the absence of tool support
    - \- standardized only for Java applications
- Performance tradeoffs