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 3 – User-defined Routines and Object Behavior

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

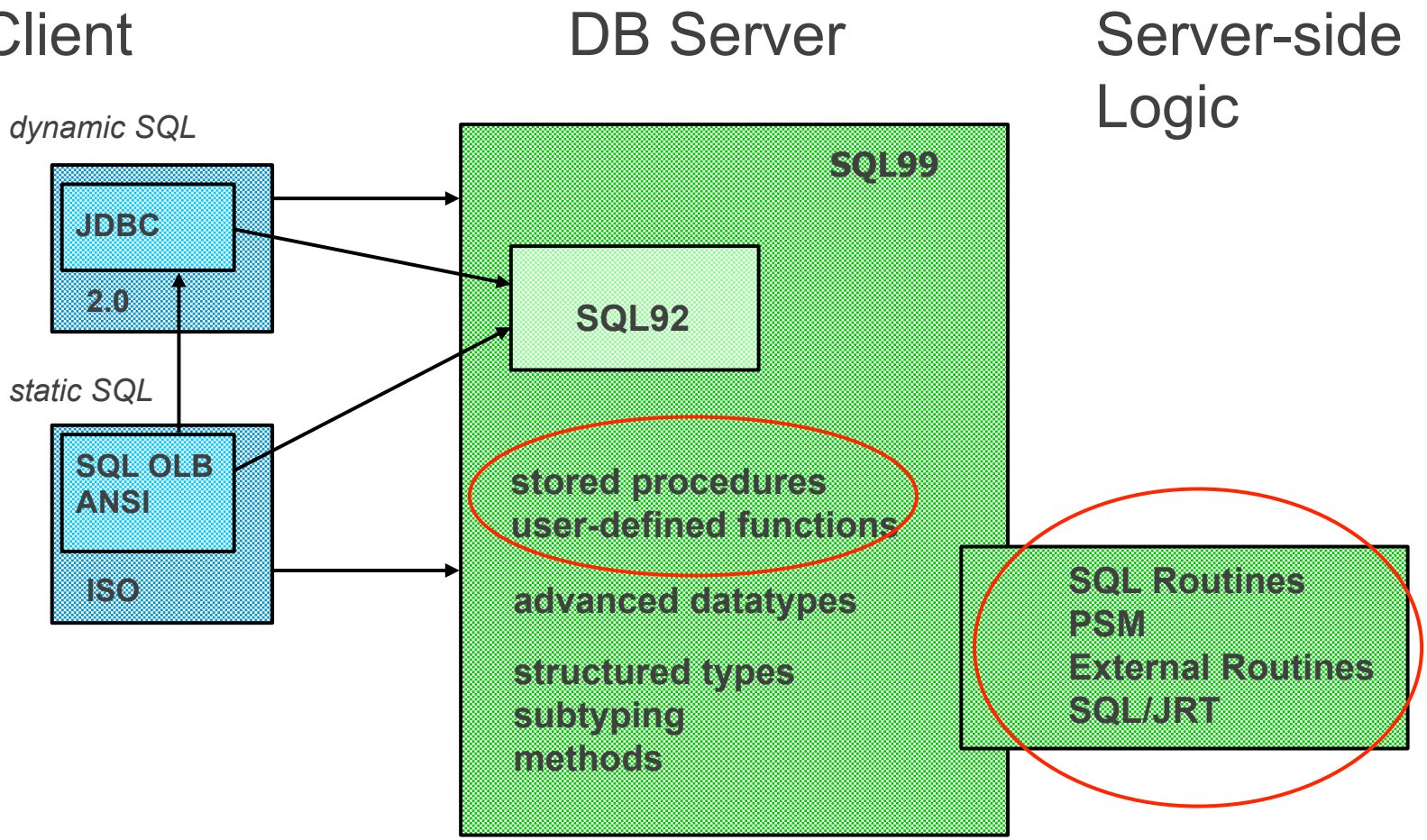
III. XML

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"



User-defined Routines

- Named persistent code to be invoked from SQL
- Can be implemented using
  - procedural SQL extensions (PSM)
  - external programming language
- Created directly in a schema
  - Have schema-qualified names
- Supported DDL
  - CREATE and DROP statements
  - ALTER statement (limited in functionality)
- Privileges
  - EXECUTE privilege controlled through GRANT and REVOKE statements
- Described by corresponding information schema views

Procedures, Functions and Methods

- Procedure
  - invoked exclusively using the SQL CALL statement
    CALL getPropertiesCloseTo ('1234 Cherry Lane …', 50, :number);
  - may return additional results in form of result sets
- Functions
  - invoked in an expressions within other SQL statements (e.g., a SELECT or UPDATE statement) using function invocation syntax
    SELECT price, location, distance(location, address('1234 Cherry Lane', …)) AS dist
    FROM properties
    ORDER BY dist
- Methods
  - are regarded as a "special kind of function", associated with structured types
  - invocation similar to function, but using method invocation syntax
    SELECT price, location, location.longitude(), location.latitude()
    FROM properties
- Reflected in additional differences in terms of
  - routine signature (header)
    - parameter mode, result
  - overloading, overriding
  - routine resolution/dispatch

Routines – General Structure

- Routine header (SQL)
  - consists of a name and a (possibly empty) list of parameters.
  - parameters of procedures may specify parameter mode
    - IN
    - OUT
    - INOUT
  - parameters of functions/methods are always IN
  - functions/methods return a single value (which may be complex)
    - header must specify data type of return value via RETURNS clause
- Routine body, specified in
  - SQL (SQL routines), using SQL procedural language extensions (PSM)
  - a host programming language (external routines)
    - may contain SQL by embedding SQL statements in host language programs or using CLI

SQL Procedural Language Extensions

Compound statement	BEGIN ... END;
SQL variable declaration	DECLARE var CHAR (6);
If statement	IF subject (var <> 'urgent') THEN ... ELSE ...;
Case statement	CASE subject (var) WHEN 'SQL' THEN ... WHEN ...;
Loop statement	LOOP < SQL statement list> END LOOP;
While statement	WHILE i<100 DO .... END WHILE;
Repeat statement	REPEAT ... UNTIL i>=100 END REPEAT;
For statement	FOR result AS ... DO ... END FOR;
Leave statement	LEAVE ...;
Return statement	RETURN 'urgent';
Call statement	CALL procedure_x (1,3,5);
Assignment statement	SET x = 'abc';
Signal/resignal statement	SIGNAL division_by_zero

Procedures

- Creating a stored procedure
  - Parameter modes OUT, INOUT designate parameters that
    - are set/modified by the procedure itself
    - are accessible by the calling application after the CALL invocation
      - mechanism depends on programming language binding

```
CREATE PROCEDURE getPropertiesCloseTo
  (IN      addr      VARCHAR(50),
   IN      distance  INTEGER,
   OUT     results   INTEGER)
...                  -- additional routine characteristics
DYNAMIC RESULT SETS 1;
```

- A procedure can return one or more result sets to a calling application
  - specified using the DYNAMIC RESULT SETS clause
  - procedure body can
    - declare cursors, indicating WITH RESULTS
    - open the cursor (to execute a SELECT statement), and leave the cursor open
  - calling application can process the results sets after invocation is completed

User-defined Functions

- Creating a user-defined function
  - parameter mode must be IN (optional)
  - must specify a result data type (RETURNS)

```
CREATE FUNCTION distance
  (loc1       VARCHAR(50),
   loc2       VARCHAR (50))
RETURNS INTEGER …;
```

- Cannot return result sets, but may issue SQL statements

User-defined Table Functions

- Motivation
  - Transform non-relational data into a relational table "on-the-fly" for further SQL processing
    - semi-structured, …
    - stored as BLOB, external file, …
    - provided by external service
      - search engine, web service, …
- Function returns a "multiset of rows"
  - Example: "docmatch" accesses an external search service, returns document ids

```
CREATE FUNCTION docmatch(idx VARCHAR(30), pattern VARCHAR(255))
  RETURNS TABLE(doc_id CHAR(16)) …;
```

- Function is used in the FROM clause of a query
  - Example: join the ids of matching documents with the DOCS table

```
SELECT T.AUTHOR, T.DOCTEXT
FROM DOCS AS T,
    TABLE(docmatch('MATHEMATICS', 'ZORN''S LEMMA')) AS F
WHERE T.DOCID = F.DOC_ID
```

Table Functions vs. Views

- View definitions don't support parameters
  - Alternative: use SQL table functions

```
CREATE FUNCTION propertiesCloseTo (loc VARCHAR(40), dist INTEGER)
  RETURNS TABLE ( price INTEGER, owner REF(person))
   ...
  RETURN TABLE(SELECT price, owner
                      FROM properties p
                      WHERE distance(p.location, loc) < dist)
```

- Use table function invocation in the FROM clause, instead of a view reference

```
SELECT *
FROM TABLE(propertiesCloseTo('1234 Cherry Lane …', 50)) AS props
WHERE props.price < 500000
```

Privilege Requirements

- Routine invocation requires EXECUTE privilege
- Whose privileges are used when the routine itself invokes SQL statements?
  - security characteristic can be specified when creating a routine
  - INVOKER
    - the invoker (user/role) must have the privileges to execute the statements in the routine body
  - DEFINER
    - the definer or creator must have the required privileges
    - routine is dropped, if the definer loses any of these privileges at a later point
    - the definer also has to possess these privileges WITH GRANT OPTION in order to grant the EXECUTE privilege to other users

Routine Overloading

- Overloading -- multiple routines with the same unqualified name

  S1.F (p1 INT, p2 REAL)

  S1.F (p1 REAL, p2 INT)

  S2.F (p1 INT, p2 REAL)

- Within the same schema
  - Every overloaded routine must have a unique signature, i.e., different number of parameters or different types for the same parameters

  S1.F (p1 INT, p2 REAL)

  S1.F (p1 REAL, p2 INT)

- Across schemas
  - Overloaded routines may have the same signature

  S1.F (p1 INT, p2 REAL)

  S2.F (p1 INT, p2 REAL)

- Functions can be overloaded by type. Procedures can only be overloaded based on number of parameters.

Subject Routine Determination

- Decides the function to invoke for a given invocation based on the
  - Compile-time data types of all arguments
  - Type precedence list of the data types of the arguments
  - SQL path
- Always succeeds in finding a unique subject function, if one exists.
- Type precedence list is a list of data type names
  - Predefined types -- defined by the standard based on increasing precision/length

    SMALLINT: SMALLINT, INTEGER, DECIMAL, NUMERIC, REAL, FLOAT, DOUBLE
    CHAR: CHAR, VARCHAR, CLOB

  - User-defined types -- determined by the subtype-supertype relationship
    - if B is a subtype of A and C is a subtype of B, then the type precedence list for C is (C, B, A).
- Invocation requires the invoker to have EXECUTE privilege on the routine -- otherwise no routine will be found for the invocation

  It is not an authorization violation!!!

Subject Routine Determination - Path

- Path is a list of schema names.
  - Can be specified during the creation of a schema, SQL-client module, or a SQL-server module

```
CREATE SCHEMA schema5
PATH schema1,schema3
...;
```

  - Every session has a default path, which can be changed using the SET statement.

```
SET PATH 'schema1, schema2'
```

Subject Routine Determination Algorithm

1. Determine the set of candidate functions for a given function invocation, F(a1, a2, ..., an):
   - Every function contained in S1 that has name F and has n parameters if the function name is fully qualified, i.e., the function invocation is of the form S1.F(a1, a2, ..., an), where S1 is a schema name.
   - Every function in every schema of the applicable path that has name F and has n parameters if the function name is not fully qualified.
2. Eliminate unsuitable candidate functions
   - The invoker has no EXECUTE privilege
   - The data type of i-th parameter of the function is not in the type precedence list of the static type of the i-th argument (for parameter)
3. Select the best match from the remaining functions
   - Examine the type of the 1st parameter of each function and keep only those functions such that the type of their 1st parameter matches best the static type of the 1st argument (i.e., occurs earliest in the type precedence list of the static type of the argument), and eliminate the rest.
   - Repeat Step b for the 2nd and subsequent parameters. Stop whenever there is only one function remaining or all parameters are considered.
4. Select the "subject function"
   - From the remaining functions take the one whose schema appears first in the applicable path (if there is only one function, then it is the "subject function")

Subject Routine Determination - Example

- Assume Y is a subtype of X. Assume the following three functions (with specific names F1, F2, and F3):

  F1: F(p1 X, p2 Y)

  F2: F(p1 Y, p2 Y)

  F3: F(p1 X, p2 REAL)

- The subject function for F(y,y) where the static type of y is Y is F2.
- Now, assume the following three functions (with specific names F4, F5, and F6):

  F4: F(p1 X, p2 Y)

  F5: F(p1 X, p2 X)

  F6: F(p1 X, p2 REAL)

- The subject function for F(y,y) where the static type of y is Y is F4.

Methods

- What are methods?
  - SQL-invoked functions "attached" to user-defined types
- How are they different from functions?
  - Implicit SELF parameter (called subject parameter)
  - Two-step creation process: signature and body specified separately.
  - Must be created in the type's schema
  - Different style of invocation, using dot-notation (e.g., *UDT-value.method*(...))

```
CREATE TYPE employee AS
(name            CHAR(40),
base_salary      DECIMAL(9,2),
bonus            DECIMAL(9,2))
INSTANTIABLE NOT FINAL
METHOD salary() RETURNS DECIMAL(9,2);

CREATE METHOD salary() FOR employee
BEGIN
....
END;
```

Methods (cont.)

- Three kinds of methods: instance, constructor, static methods
- Two types of instance methods:
  - Original methods: methods attached to (super) type
  - Overriding methods: methods attached to subtypes, redefining original behavior
    - Signature must match with the signature of an original method, except for the subject parameter

```
CREATE TYPE employee AS
(name            CHAR(40),
base_salary      DECIMAL(9,2),
bonus                        DECIMAL(9,2))
INSTANTIABLE NOT FINAL
METHOD salary() RETURNS DECIMAL(9,2);

CREATE TYPE manager UNDER employee AS
(stock_option    INTEGER)
INSTANTIABLE NOT FINAL
OVERRIDING METHOD salary() RETURNS DECIMAL(9,2),      -- overriding
METHOD vested() RETURNS INTEGER                       -- original;
```

Instance Methods

- Invoked using dot syntax (assume dept table has mgr column):

  SELECT mgr.salary() FROM dept;
- Subject routine determination picks the "best" method to invoke.
  - Same algorithm as used for regular functions
  - SQL path is temporarily set to a list with the schemas of the supertypes of the static type of the self argument.
- Dynamic dispatch executed at runtime
  - Overriding methods considered at execution time
  - Overriding method with the best match for the dynamic type of the self argument is selected.
  - Schema evolution affects the actual method that gets invoked. If there is a new overriding method defined it may be picked for execution.

Method Reference

- References can be used to invoke methods on the corresponding structured type
  - assumption: type 'person' has a method 'income' with the appropriate signature

    ```
    SELECT prop.price, prop.owner->income(1998)
    FROM properties prop
    ```

    ```
    SELECT name, DEREF(oid).income(1998)
    FROM people
    ```
- Invocation of methods given a reference value require select privilege on the method for the target typed table

  ```
  GRANT SELECT (METHOD income FOR person) ON TABLE people TO PUBLIC
  ```
  - Allows the table owner to control who is authorized to invoke methods on the rows of his/her table

Manipulating Structured Type Attributes

- Attributes of structured types are implicitly associated with a pair of instance methods
- **Observer** and **mutator** methods are used to access and modify attributes
  - Automatically generated when type is defined
    CREATE TYPE address AS (street CHAR (30), city CHAR (20), state CHAR (2), zip INTEGER) NOT FINAL

address_expression.street () -> CHAR (30)
address_expression.city () -> CHAR (20)
address_expression.state () -> CHAR (2)
address_expression.zip () -> INTEGER
address_expression.street (CHAR (30)) -> address
address_expression.city (CHAR (20)) -> address
address_expression.state (CHAR (2)) -> address
address_expression.zip (INTEGER) -> address

Dot Notation

- "Dot" notation must be used to invoke methods (e.g., to access attributes)
- Methods without parameters do not require use of "()"
  - SELECT location.street, location.city( ), location.state, location.zip( )
    FROM properties
    WHERE price < 100000
- Support for several 'levels' of dot notation (a.b.c.d.e)
- Allow "navigational" access to structured types
- Support for "user-friendly" assignment syntax
  - DECLARE r real_estate;
  - ...
  - SET r.size = 2540.50; -- same as r.size (2540.50)
  - ...
  - SET ... = r.location.state; -- same as r.location( ).state( )
  - SET r.location.city = `LA'; -- same as r.location(r.location.city(`LA'))
- Dot notation does not 'reveal' physical representation
  - allows the definition of 'derived' attributes
    - method 'longitude' is indistinguishable from attribute 'city' from an invocation perspective

Encapsulation

- An object should encapsulate its state to the outside world
  - only the methods of an object may access the object state directly
  - other objects must invoke interface methods, cannot directly access the state
  - separate interface from implementation
- Encapsulation through public interface definition (not supported by SQL!)
  - strict
    - all attributes are encapsulated
    - all or subset of methods are part of the interface
  - flexible
    - individual attributes and methods may be designated as private, public, protected
- Encapsulation implemented through privileges (supported by SQL)
  - use authorization concepts to achieve arbitrary 'levels' of encapsulation
  - EXECUTE privilege may be granted/revoked on observer/mutator methods as well

Static Methods

- Static Methods
  - have no subject (SELF) parameter
  - behavior associated with type, not instance
  - no overriding, dynamic dispatch
- Created using keyword STATIC

```
CREATE TYPE employee ...
STATIC METHOD totalSalary(base DECIMAL(9,2), bonus DECIMAL(9,2))
    RETURNS DECIMAL(9,2);
```

- Invocation uses structured type name, "::"
  - syntax 'borrowed' from C++

```
VALUES (employee::totalSalary(70000, 10000));
```

Initializing Instances: Constructor

- Instances are generated by the system-provided constructor function
  - Attributes are initialized with their default values
- Attributes are modified (further initialized) by invoking the mutator functions

```
BEGIN
    DECLARE re real_estate;
    SET re = real_estate();                 -- generation of a new instance
    SET re.rooms = 12;                      -- initialization of attribute rooms
    SET re.size = 2500;                     -- initialization of attribute size
END

BEGIN
    DECLARE re real_estate;
    SET re = real_estate().rooms (12).size (2500);  -- same as above
END
```

User-defined Constructor Methods

- Users can define any number of constructor methods and invoke them with NEW operator

```
CREATE TYPE real_estate AS ( ....)
CONSTRUCTOR METHOD real_estate (r INTEGER, s DECIMAL(8,2)) RETURNS real_estate

CREATE CONSTRUCTOR METHOD real_estate
       (r INTEGER, s DECIMAL(8,2)) RETURNS real_estate FOR real_estate
BEGIN
    SET self.rooms = r;
    SET self.size = s;
    RETURN self;
END

BEGIN
    DECLARE re real_estate;
    SET re = NEW real_estate(12, 2500);                    -- same as previously
END
```

Methods That Modify Object State

- In OO-programming
  - a method that wants to modify the state of its object simply assigns new values to an attribute
  - changes are reflected in the identical object
- In SQL
  - value-based operations
    - expressions (including method invocations) always return (copies of) values
    - persistent data can only be updated by the respective DML operations (e.g., UPDATE), assigning the results of expressions to the columns to be modified
  - a method will always operate on a copy of a complex value (i.e., instance of a structured type)
    - modification of state as a pure side-effect of a method is not possible
  - modifying the object state will require an UPDATE statement
    - method returns a modified copy of the original complex value
    - separate UPDATE replaces old value with the new copy

Value-based Model in SQL

- SQL functions/methods operate on a value-based model
  - method needs to return a modified copy of the SELF object
    - return type must be the ST

```
CREATE TYPE employee AS
(...)
INSTANTIABLE NOT FINAL
METHOD salary (DECIMAL(9, 2)) RETURNS employee;

CREATE METHOD salary (newsal DECIMAL(9, 2)) FOR employee
BEGIN
    SET self.base_salary = newsal;
    RETURN self;
END;

UPDATE Employees            -- assumes the table has an emp column of type employee!
    SET emp = emp.salary(80000)
    WHERE emp.name = 'Smith';

UPDATE Employees
    SET emp.salary = 80000              -- same as above
    WHERE emp.name = 'Smith'
```

Substitutability and Value-based Model

- Problems with static type checking

```
CREATE TYPE employee AS (...)
    INSTANTIABLE NOT FINAL
    METHOD salary (DECIMAL(9, 2)) RETURNS employee;
CREATE TYPE manager UNDER employee AS (...)
    INSTANTIABLE NOT FINAL;
CREATE TABLE departments
    (..., mgr manager, ...)

UPDATE departments
    SET mgr = mgr.salary(80000)
    WHERE depno = 'K55';
```

manager inherts method salary

method salary has static result type 'employee';
but department.mgr has type 'manager'!

=> static type error!

Type-Preserving Functions/Methods

- SQL-invoked function, one of whose parameters is a **result SQL parameter**.
  - The most specific type of the value returned by an invocation of a type-preserving function is identical to the most specific type of the SQL argument value substituted for the result SQL parameter
  - This can be the SELF parameter for methods
- Example:

```
CREATE TYPE employee AS
(…)
INSTANTIABLE NOT FINAL
METHOD salary (DECIMAL(9, 2)) RETURNS employee SELF AS RESULT;


UPDATE departments
    SET mgr = mgr.salary(80000)
    WHERE depno = 'K55';
```

  Type-checking succeeds, although the return type of manager.salary() is **employee**!
- All system-generated mutator methods are type-preserving

Additional Routine Characteristics

- DETERMINISTIC or NOT DETERMINISTIC
  - DETERMINISTIC (default)
    - Routine is expected to return the same result/output values for a given list of input values. (However, no checks are done at run time.)
    - Gives the SQL query engine full flexibility for rewrite and optimization purposes
  - NOT DETERMINISTIC routines not allowed in
    - Constraint definitions
    - Assertions
    - In the condition part of CASE expressions
    - CASE statements
- RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT (default)
  - RETURNS NULL ON NULL INPUT
    - An invocation returns null result/output value if any of the input values is null without executing the routine body

Additional Routine Characteristics (cont.)

- CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA
  - External routines may in addition specify NO SQL
  - Implementation-defined default
  - For SQL routines -- check may be done at routine creation time
  - For both SQL and external routines -- exception raised if a routine attempts to perform actions that violate the specified characteristic
  - Routines with MODIFIES SQL DATA not allowed in
    - Constraint definitions
    - Assertions
    - Query expressions (SELECT …)
    - Triggered actions of BEFORE triggers
    - Condition part of CASE expressions
    - CASE statements
    - searched delete statements
    - search condition of searched update statements (are allowed in SET clause)

External Routines

- Motivation
  - external programming language may have better performance for computationally extensive tasks
  - leverage existing program libraries
- CREATE statement does not contain a method body
  - LANGUAGE clause
    - Identifies the host language in which the body is written
  - NAME clause
    - Identifies the host language code, e.g., file path in Unix

```
CREATE FUNCTION get_balance( IN INTEGER) RETURNS DECIMAL(15,2))
LANGUAGE C EXTERNAL NAME 'usr/McKnight/banking/balance'
```

External Routine Parameters and Implementation

- Parameters
  - Names are optional
    - are not used in the routine body
  - Permissible data types depend on the host language of the body
- RETURNS clause may specify **CAST FROM** clause

```
CREATE FUNCTION get_balance( IN INT)
RETURNS DECIMAL(15,2)) CAST FROM REAL
LANGUAGE C
```

  - C program returns a REAL value, which is then cast to DECIMAL(15,2) before returning to the caller.
- Parameter Styles
  - Define the signature of the corresponding programming language routine
  - Special provisions to handle **null indicators** and the **status** of execution (SQLSTATE)
    - PARAMETER STYLE SQL (is the default)
    - PARAMETER STYLE GENERAL

PARAMETER STYLE SQL

- Additional parameters necessary for null indicators and routine name, and for returning a function result, error message, and SQLSTATE value
- External language program (i.e., the body) has 2n+4 parameters for procedures and 2n+6 parameters for functions where n is the number of parameters of the external routine

```
CREATE FUNCTION get_balance( IN INTEGER)
RETURNS DECIMAL(15,2)) CAST FROM REAL
LANGUAGE C
EXTERNAL NAME 'bank\balance'
PARAMETER STYLE SQL
```

```
void balance (int* acct_id,
float* rtn_val,
int* acct_id_ind,
int* rtn_ind,
char* sqlstate[6],
char* rtn_name [512],
char* spc_name [512],
char* msg_text[512])
{
...
}
```

PARAMETER STYLE GENERAL

- No additional parameters
- External language program (i.e., the body) must have exactly the same number of parameters
- Cannot handle null values
  - Exception is raised if any of the arguments evaluate to null
- Value is returned in an implementation-dependent manner

```
CREATE FUNCTION get_balance( IN INTEGER)
RETURNS DECIMAL(15,2)) CAST FROM REAL
LANGUAGE C
EXTERNAL NAME  'bank\balance'
PARAMETER STYLE GENERAL
```

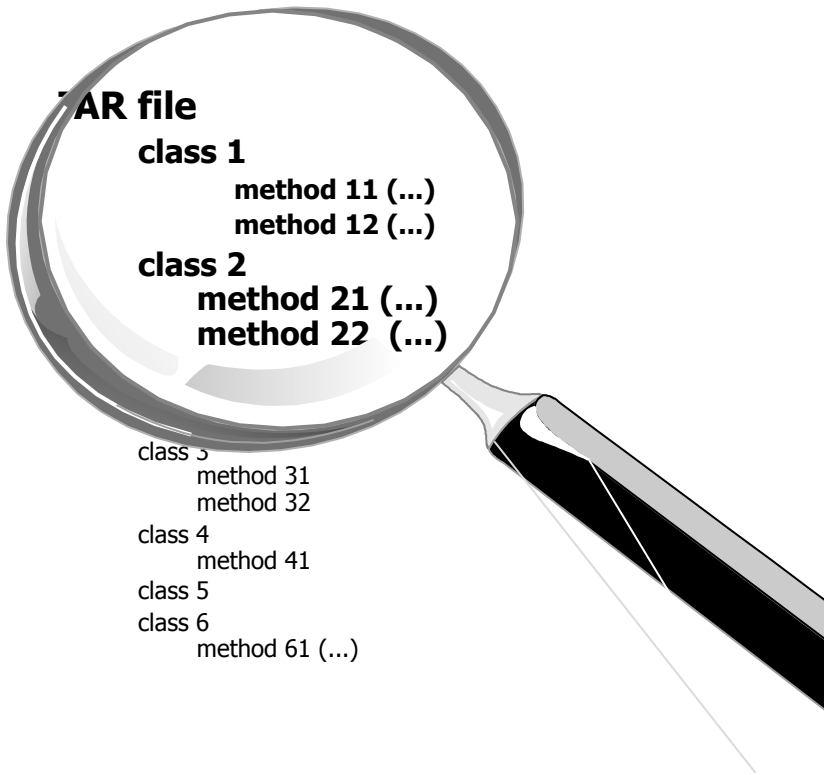
```
float* balance (int* acct_id)
{
...
}
```

SQLJ Part 1

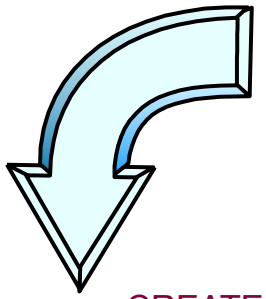
- SQL Routines using the Java™ Programming Language
  - LANGUAGE JAVA PARAMETER STYLE JAVA
    - no additional parameters on Java method signatures required to handle null values, errors, etc.
  - Java static methods used to implement SQL stored procedures and user-defined functions
    - parameter type 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 as SQL/JRT

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')

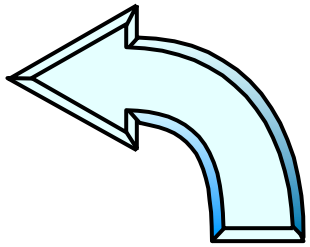


Creating Procedures and UDFs

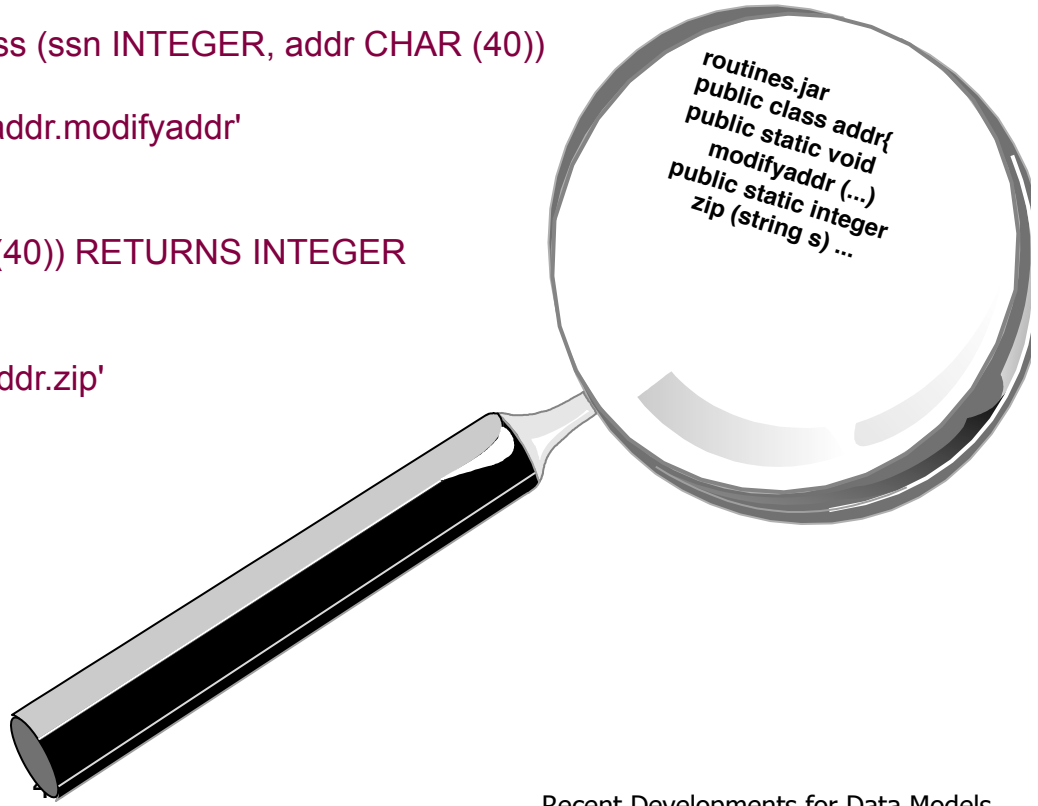


sqlj.install_jar ('file:~/classes/routines.jar', 'routines_jar')

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
```



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 structured types
- Mapping of object **state** and **behavior**
  - Java methods become SQL 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
- Uses 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 as SQL/JRT

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'
)
```

Instance Update Methods

- Remember: Java and SQL have different object update models
  - Java model is object-based
  - SQL model is value-based
- SQLJ permits mapping without requiring modification of Java methods
  - SELF AS RESULT 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*

Summary

- OODBMS features
  - Extensibility
    - user-defined types (structure and **operations**) as first class citizens
  - Computational completeness
    - use DML to express any computable function (-> method implementation)
  - Encapsulation
    - separate specification (interface) from implementation
  - Overloading, overriding, late binding
    - same name for different operations or implementations
- SQL
  - User-defined routines
    - procedures, functions, methods (instance, static, constructor)
  - SQL routines and external routines
    - use SQL/PSM procedural extensions or external PL to implement routine
  - Encapsulation
    - observer/mutator methods and EXECUTE privilege to achieve encapsulation
  - Overloading
    - full support for functions, methods
    - limited support for procedures
  - Overriding, late binding
    - supported for methods