Part X

Application Programming
Application Programming

1. Programming Language Connection
2. JDBC
3. SQLJ
4. LINQ
5. Object-Relational Mapping
6. Procedural SQL-Extensions: SQL/PSM
Learning goals for today . . .

- Knowledge about concepts and interfaces for access on SQL-databases out of programming languages
- Understanding of procedural interfaces on the example of JDBC
- Knowledge on embedded SQL and procedural SQL-extensions
- Basic knowledge on object-relational mapping
Programming Language Connection

- **Coupling types:**
  - Procedural or CALL-interfaces (*call level interface*)
    - Examples: SQL/CLI, ODBC, JDBC, …
  - Embedding of a DB-language into programming languages
    - Static embedding: Precompiler-principle
      - SQL-Statements defined *at compile time*
    - Examples: Embedded SQL, SQLJ
    - Dynamic embedding:
      - construction of SQL-statements at runtime
  - **Language extensions** and new *language developments*
    - Examples: SQL/PSM, PL/SQL, Transact-SQL, PL/pgSQL
Cursor-Concept

- **Cursor**: iterator over list of tuples (query result)
JDBC: Overview

- Database access interface for Java
- Abstract, database neutral interface
- Comparable with ODBC
- Low-Level-API: direct usage of SQL
- Java-Package `java.sql`
  - `DriverManager`: Entrance point, loading of drivers
  - `Connection`: Database connection
  - `Statement`: Execution of statement with a connection
  - `ResultSet`: Manages results of a query, access on single columns
JDBC: Structure

1. DriverManager
   - getConnection

2. Connection
   - createStatement
   - executeQuery

3. Statement
   - ResultSet

4. Statement
   - ResultSet
JDBC: Driver Concept

![Diagram of JDBC Driver Concept]

- Java Application
- JDBC Driver Manager
  - Native Protokoll Driver
  - JDBC Net Driver
  - JDBC ODBC Bridge
  - Native API Driver
- DB Middleware
- ODBC
- Client Library
JDBC: Sequence of Events

1. Establishing of a connection to the database
   - Specification of connection information
   - Selection and loading of the driver

2. Sending of a SQL-query
   - Definition of the statement
   - Assignment of parameters

3. Processing of the query results
   - Navigation over result relation
   - Access on columns
JDBC: Connection Establishment

1. Loading drivers

   ```java
   Class.forName("com.company.DBDriver");
   ```

2. Establish connection

   ```java
   String url = "jdbc:subprotocol:datasource";
   Connection con = DriverManager.getConnection(url, "scott", "tiger");
   ```

JDBC-URL specifies

- Data source / Database
- Connection mechanism (Protocol, Server and Port)
JDBC: Query Execution

1. Create statement

```java
Statement stmt = con.createStatement();
```

2. Execute statement

```java
String query = "select Name, Vintage from WINES";
ResultSet rSet = stmt.executeQuery(query);
```

Class `java.sql.Statement`

- Execution of queries (SELECT) with `executeQuery`
- Execution of changing statements (DELETE, INSERT, UPDATE) with `executeUpdate`
JDBC: Result Processing

1 Navigation over result set (Cursor-Principle)

```java
while (rSet.next()) {
    // Processing of single tuples
    ...
}
```

2 Access of column values with `getType`-methods

- with column index

```java
String wName = rSet.getString(1);
```

- with column name

```java
String wName = rSet.getString("Name");
```
JDBC: Exception Handling

- Exception handling with **try-catch-mechanism**
- SQLException for all SQL- and DBMS-exceptions

```java
try {
    // call of JDBC-methods
    ...
} catch (SQLException exc) {
    System.out.println("SQLException: " +
        exc.getMessage());
}
```
JDBC: Update Operations

- DDL- and DML-statements with `executeUpdate`
- Gives number of affected rows (for DML-statements)

```java
Statement stmt = con.createStatement();
int rows = stmt.executeUpdate(
    "update WINES set Price = Price * 1.1 " +
    "where Vintage < 2000");
```
JDBC: Transaction Management

- **Methods of Connection**
  - commit()
  - rollback()

- **Auto-Commit-Mode**
  - Implicit commit after each statement
  - Transaction consists just out of one single statement
  - Switch mode with setAutoCommit(boolean)
SQLJ: Embedded SQL for Java

- Embedding of SQL-statements in Java source code
- Precompilation of the extended source codes onto real Java code with the translator `sqlj`
- Checking of the SQL-statements
  - Correct syntax
  - Accordance of the statements with the DB-scheme
  - Type compatibility of the for data transfer used variables
- Usage of JDBC-drivers
SQLJ: Principle

SQLJ Program

SQLJ Translator

Syntax & Semantic Check

Java Source Code

Java Compiler

Bytecode

Java Compiler

SQLJ Profiles

Customizer

Custom Profile

SQLJ Runtime System

JDBC Driver

Saake

Database Concepts

Last Edited: June 2019
SQLJ-Statements

- Identification with #sql declaration
- Class definition for iterators
- SQL-statements: Queries, DML- and DDL-statements

```sql
#sql { SQL-statement };
```

Example:

```sql
#sql { insert into PRODUCER (Vineyard, Region) values
    ( 'Wairau Hills', 'Marlborough' ) };
```
Host-Variables

- Variables of a host-language (here Java) that can occur in SQL-statements
- Usage: Exchange of data between the host-language and SQL
- Identification with ":variable"

Example:

```java
String name;
int wineID = 4711;
#sql { select Name into :name
    from WINES where WineID = :wineID }
System.out.println("Wine = " + name);
```
Iterators

1. **Declaration of the iterator**

   ```java
   #sql public iterator WineIter(String Name, String Vineyard, int Vintage);
   ```

2. **Definition of the iterator object**

   ```java
   WineIter iter;
   ```

3. **Execution of the statement**

   ```java
   #sql iter = { select Name, Vineyard, Vintage from WINES };
   ```

4. **Navigation**

   ```java
   while (iter.next()) {
       System.out.println(iter.Name() + " " + iter.Vineyard() + " " + iter.Vintage());
   }
   ```
Dynamic SQL

- SQL-Statements as during runtime constructed Strings

```sql
exec sql begin declare section;
    QueryString char(256) varying;
exec sql end declare section;
exec sql declare QueryObjekt statement;
QueryString =
    'delete from WINES where WineID = 4711';
...
exec sql prepare QueryObjekt from :QueryString;
exec sql execute QueryObjekt;
```
Language Integrated Query (LINQ)

- Embedding of a DB-language (SQL) into a programming language (C#)
- Specialized class methods

```csharp
IEnumerable<string> res = wines
    .Where(w => w.Color == "Red")
    .Select(w => new { w.Name });
```

- Own language constructs (since C# 3.0)

```csharp
IEnumerable<op> res = from w in wines
    where w.Color == "Red"
    select new { w.Name };
Object-Relational Mapping

- Use of
  - Relational back ends (SQL-DBMS)
  - Object-relational applications, applications servers, middle ware, ...

- Implementation of "‘business logic’" in form of objects (customer, order, process, ...)  
  - e.g., as Java Bean, CORBA-object

- Requires: Mapping class $\leftrightarrow$ relation

- Aspects:
  - Conceptual mapping
  - Runtime support

- Technologies/Products: JDO, Hibernate, ADO.NET Entity Framework. . .
Object-Relational Mapping: Principle
Classes and Tables

- **OO**: Class defines properties of objects (intention) + covers set of all objects (extension)
- **RM**: Relation covers all tuples, relational scheme describes structure
- **Obvious**: class = table
- **But**: normalization decomposes relations!
  - 1 class = 1 table
  - 1 class = $n$ tables
  - $n$ classes = 1 table
Classes and Tables: Example

Wine
- Id : int
- Name : string
- Color : string
- Vintage : integer
- Vineyard : string

Wine
- WineID
- Name
- Color
- Vintage
- Vineyard
Relations

- **Embedded foreign key** in the relation of the class, i.e. the identifier of the associated object is saved as foreign key in additional columns
- **Foreign key tables**: the relation instance is represented as tuple with the keys of the involved objects
- Mapping of the relating classes on a **single table**: violation of the normal form
- Concrete
  - 1:1-Relation: embedded foreign keys
  - 1:n-Relation: embedded foreign keys of foreign key tables
  - Relations with attributes: Foreign key tables
  - m:n-Relations: Foreign key tables
  - Three- and more valued relations: Foreign key tables
Relations /2

- **Producer**
  - Name : string
  - District : string
  - Region : string
  - Winemaker: list of string

- **Producer**
  - Vineyard
  - District
  - Region

- **Winemaker**
  - Vineyard
  - Name

The diagram shows the relationships between Producers, Vineyards, Districts, Regions, and Winemakers.
Hibernate

- Java-framework for object-relational mapping
- Idea: Mapping of Java-objects to tuples of a relational database
- Principle: Java-class + mapping rule $\rightarrow$ SQL-table
- No explicit SQL-statements required!
- Support of the navigation over relations (automatic loading of the referenced objects)
- Queries on some languages (HQL resp. QBC/QBE)
Hibernate: Example

```java
public class Wine {
    private int id;
    private String name;
    private String color;
    private int vintage;
    private String vineyard;

    public void setName(String n) { name = n; }
    public String getName() { return name; }
    public void setColor(String c) { color = c; }
    public String getColor() { return color; }
    public void setVintage(int v) { vintage = v; }
    public int getVintage() { return vintage; }
    ...
}
```
Hibernate: Example /2

- Declaration of the mapping in a XML-Mapping-File
- Mapping rule is interpreted during runtime

```xml
<hibernate-mapping>
  <class name="Wine" table="WINES">
    <id name="id">
      <generator class="native" />
    </id>
    <property name="name" />
    <property name="color" />
    <property name="vintage" column="vintage" />
    <property name="vineyard" />
  </class>
</hibernate-mapping>
```
Hibernate: Object Creation

```java
Transaction tx = null;

Wine wine = new Wine();
wine.setName("Pinot Noir");
wine.setColor("Red");
wine.setVintage(1999);
wine.setVineyard("Helena");

try {
    tx = session.beginTransaction();
    session.save(wine);
    tx.commit();
} catch (HibernateException exc) {
    if (tx != null) tx.rollback();
}
```
Queries with Hibernate’s query language HQL

Formulation on the *conceptual* scheme (Java-classes)

Select-clause not required (results are always objects)

Example

```java
Query query =
    session.createQuery("from Wine where Color = 'Red'");
Iterator iter = query.iterate();
while (iter.hasNext()) {
    Wine wine = (Wine) iter.next();
    ...
}
```
SQL/PSM: The Standard

- SQL-Standard for procedural extensions
- PSM: Persistent Stored Modules
  - Stored modules of procedures and functions
  - Single routines
  - Integration of external routines (implemented in C, Java, ...)  
  - Syntactic constructs for loops, conditions etc.
  - Basis for method implementation for object-relational concepts
Advantages of Stored Procedures

- Proved structuring tool for larger applications
- Specification of functions and procedures done in the database language; thus only depending on DBMS
- Optimization by DBMS possible
- Execution of the procedures completely under control of the DBMS
- Central control of the procedures allows a redundancy free representation of relevant aspects of the application functionality
- Concepts and mechanisms of the right assignment of the DBMS can be extended on procedures
- Procedures can be used for integrity protection (e.g., as action part of triggers)
SQL/PSM: Variable Declaration

- Declare variables before consumption
- Specification of identifier and data type
- Optional with initial value

```sql
declare Price float;
declare Name varchar(50);
declare Set int default 0;
```
SQL/PSM: Flow Control

- Assignment

```sql
set var = 42;
```

- Conditional branching

```sql
if <Condition> then <Statement>
[ else <Statement> ] end if;
```
Loops

loop <Statement> end loop;
while <Condition> do
  <Statement> end while;
repeat <Statement>
  until <Condition> end repeat;
Loops with cursor

```
for LoopVariable as CursorName cursor for
  CursorDeclaration
  do
    Statement
  end;
```
**SQL/PSM: Flow Control**

```sql
declare wlist varchar(500) default '';
declare pos integer default 0;

for w as WineCurs cursor for
    select Name from WINES where Vintage = 'Helena'
do
    if pos > 0 then
        set wlist = wlist || ',' || w.Name;
    else
        set wlist = w.Name;
    end if;
    set pos = pos + 1;
end for;
```
SQL/PSM: Exception Handling

- Triggering of an exception (Condition)

```sql
signal <ConditionName>;
```

- Declaration of exceptions

```sql
declare missing_vineyard condition;
declare invalid_region
  condition for sqlstate value '40123';
```
Exception handling

```sql
begin
    declare exit handler for ConditionName
    begin
        -- statements for exception handling
    end
    -- statements that can trigger exceptions
end
```
Function definition

```sql
create function taste (rz int) returns varchar(20)
begin
    return case
        when rz <= 9 then 'Dry'
        when rz > 9 and rz <= 18 then 'Medium-Dry'
        when rz > 18 and rz <= 45 then 'Smooth'
        else 'Sweet'
    end
end
```
SQL/PSM: Functions /2

- Call inside of a query

```sql
select Name, Vineyard, taste(residualSugar)
from WINES
where Color = 'Red' and taste(residualSugar) = 'Dry'
```

- usage outside of queries

```sql
set wine_taste = taste(12);
```
Procedure definition

```sql
create procedure winelist (in prod varchar(30),
                         out wlist varchar(500))
begin
    declare pos integer default 0;

    for w as WineCurs cursor for
        select Name from WINES where Vintage = prod
    do
        -- see example of slide 10-38
    end for;
end; end;
```
SQL/PSM: Procedures /2

- Usage via `call`-statement

```sql
declare wlist varchar(500);
call winelist ('Helena', wlist);
```
SQL/PSM: Access Characteristics

Properties of procedures that affect query execution and optimization

- **Deterministic**: Routine gives same results for same parameters
- **No SQL**: Routine contains no SQL-statements
- **Contains SQL**: Routine contains SQL-statements (standard for SQL-routines)
- **Reads SQL data**: Routine executes SQL-queries (select-statements)
- **Modifies SQL data**: Routine that contains DML-statements (insert, update, delete)
Control Questions

- What concepts exist that can access SQL-databases?
- What are advantages and disadvantages of call-level-interfaces such as JDBC in comparison with embedding of SQL?
- How can application objects be mapped to SQL-tables? What tasks are therefore required?
Summary

- Connection between SQL and imperative languages
- Call-level-interfaces vs. embedded SQL
- Object relational mapping
- SQL/PSM: imperative extension of SQL → implementation of functions and procedures