Last lecture

We started with querying in SQL.

Basic Query Structure

SELECT <attribute list>
FROM <table list>
WHERE <condition>;

where

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

Simple Example

Query 0. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
Q0:   SELECT   Bdate, Address
      FROM     EMPLOYEE
      WHERE    Fname='John' AND Minit='B' AND Lname='Smith';
```

Basic Syntax of a JOIN

Most simple syntax is just listing multiple tables in FROM clause:

```
FROM Employee, Department
```

Note that this is equivalent to the **Carthesian Product A x B**, which we said is usually not very useful

Usually we combine the JOIN with **explicitly linking** the tables through their PK/FKs

```
SELECT E.Fname, E.Lname, D.Dname
FROM Employee as E, Department as D
WHERE E.Dno = D.Dnumber;
```

Outer Joins

SQL also supports the LEFT, RIGHT, and FULL style of outer joins

```
SELECT  E.Fname, E.Lname, S.Fname, S.Lname
FROM  EMPLOYEE AS E LEFT OUTER JOIN
   EMPLOYEE AS S ON E.Super_ssn=S.Ssn;
```

RIGHT and FULL outer joins follow in the same style

In outer joins we are **required to use** the “On” syntax

Unions, Intersect, Except

Finally, SQL also supports the set operators UNION, INTERSECT, and EXCEPT (difference)

Query 4. Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

```
Q4A:  ( SELECT    DISTINCT Pnumber
        FROM      PROJECT, DEPARTMENT, EMPLOYEE
        WHERE     Dnum=Dnumber AND Mgr_ssn=Ssn
                  AND Lname='Smith' )
        UNION
      ( SELECT    DISTINCT Pnumber
        FROM      PROJECT, WORKS_ON, EMPLOYEE
        WHERE     Pnumber=Pno AND Essn=Ssn
                  AND Lname='Smith' );
```

Subqueries

Some queries require us to execute **another nested query** as part of the WHERE clause

Typically in conjunction with **predicate logic operators:**

IN, ALL, ANY (SOME), EXISTS, UNIQUE

Find all employees with higher salary than all employees in department number 5:

```
SELECT   Lname, Fname
FROM     EMPLOYEE
WHERE    Salary > ALL  ( SELECT   Salary
                         FROM     EMPLOYEE
                         WHERE    Dno=5 );
```

“WITH” Subqueries

Sometimes it is easier to just create a temporary table instead of doing a “true” subquery:

```
WITH dno_5_sales as (
     SELECT salary from EMPLOYEE WHERE Dno = 5)

  SELECT Lname, Fname FROM EMPLOYEE
  WHERE Salary > ALL (dno_5_sales);
```

Subqueries

```
Q4A:  SELECT   DISTINCT Pnumber
      FROM     PROJECT
      WHERE    Pnumber IN
               ( SELECT     Pnumber
                 FROM       PROJECT, DEPARTMENT, EMPLOYEE
                 WHERE      Dnum=Dnumber AND
                            Mgr_ssn=Ssn AND Lname='Smith' )
               OR
               Pnumber IN
               ( SELECT     Pno
                 FROM       WORKS_ON, EMPLOYEE
                 WHERE      Essn=Ssn AND Lname='Smith' );
```

Using tables from the superquery in the subquery

Often is is required to reference something (tables, attributes, etc.) from the superquery in the subquery

Needs aliases

Query 16. Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
Q16:  SELECT   E.Fname, E.Lname
      FROM     EMPLOYEE AS E
      WHERE    E.Ssn IN ( SELECT   Essn
                          FROM     DEPENDENT AS D
                          WHERE    E.Fname=D.Dependent_name
                          AND E.Sex=D.Sex );
```

Aggregate Functions

Used to summarize information from multiple tuples into a single-tuple summary

Available aggregate functions:

`COUNT`, `SUM`, `MAX`, `MIN`, and `AVG`

NULL values are discarded when aggregating

Examples

Return a single row with summary statistics of employees:

```
SELECT SUM(Salary), MAX(Salary),
  MIN(Salary), AVG(Salary) FROM  EMPLOYEE;
```

Often used in combination with aliasing:

```
SELECT SUM(Salary) AS Total_Sal,
  MAX(Salary) AS Highest_Sal,
  MIN(Salary) AS Lowest_Sal,
  AVG(Salary) AS Average_Sal
FROM  EMPLOYEE;
```

Grouping

Common problem when aggregating:

We don't want to get the aggregate of **all** rows, but somehow group them by a specific attribute

(**partition** relation into subsets of tuples)

Example: find the average salary **per department**

Solution: GROUP BY clause

Grouping

Example: get the number of employees and their average salary **per department**

```
SELECT  Dno, COUNT(*), AVG (Salary)
  FROM  EMPLOYEE
  GROUP BY Dno;
```

Unlike other aggregate functions, this query **does not** return just one row; it returns one row **per distinct grouping value** (in that case department numbers).

Basic operating principle:

First apply the grouping (i.e., figure out how many rows the result will have)

Then apply the aggregate functions for each group

Grouping

Important restrictions:

- the grouping attribute(s) **must** appear in the SELECT clause
- other non-aggregate attribute(s) **cannot** appear in the SELECT clause

Wrong:

```
SELECT  COUNT(*), AVG (Salary)
  FROM  EMPLOYEE
  GROUP BY Dno;
```

```
SELECT  COUNT(*), AVG (Salary), Salary
  FROM  EMPLOYEE
  GROUP BY Dno;
```

Grouping and WHERE

Grouping may also be done in combination with a WHERE clause, for instance as part of a JOIN

```
SELECT  Pnumber, Pname, COUNT(*)
  FROM  PROJECT, WORKS_ON
  WHERE  Pnumber=Pno
  GROUP BY  Pnumber, Pname;
```

Note that the WHERE clause is evaluated **before** the grouping happens.

Filtering after grouping

Sometimes we need to reject (filter out) an entire group - this cannot be done with WHERE

Use HAVING:

```
SELECT  Pnumber, Pname, COUNT(*)
  FROM  PROJECT, WORKS_ON
  WHERE  Pnumber=Pno
  GROUP BY  Pnumber, Pname
  HAVING  COUNT(*) > 2;
```

Kahoot! Quiz

Let’s do some small examples.

Basic SQL Query Syntax (order matters!)

```
[WITH <name> AS <subquery>]
SELECT <attribute and function list>
FROM <table list>
[WHERE <single condition>]
[GROUP BY <grouping attributes>]
[HAVING <group condition>]
[ORDER BY <attribute list];
```

Processing order:

```
1:   [WITH <name> AS <subquery>]
6:   SELECT <attribute and function list>
2:   FROM <table list>
3:   [WHERE <single condition>]
4:   [GROUP BY <grouping attributes>]
5:   [HAVING <group condition>]
7:   [ORDER BY <attribute list];
```

Key Takeaways for SQL

Understanding basic SQL queries (SELECT, FROM, WHERE)

DISTINCT, ORDER BY

Different types of joins and their syntax

Subselects and WITH

Aggregation, Grouping, and the HAVING clause

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Interacting with the Database from Java

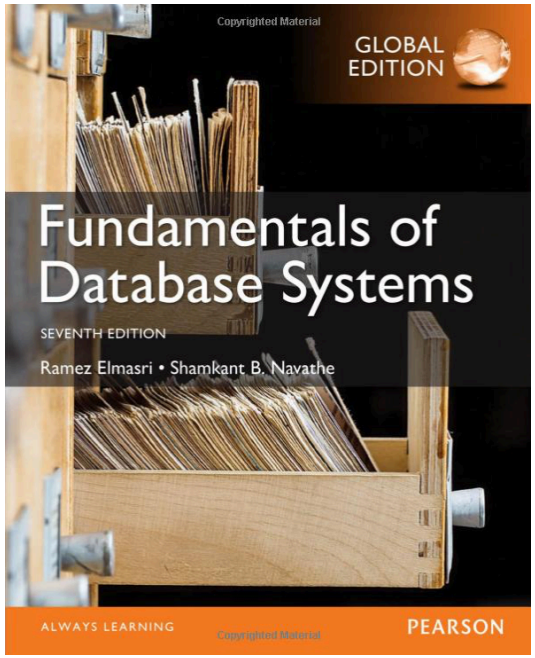
LECTURE 9

Dr. Philipp Leitner

philipp.leitner@chalmers.se

@xLeitix

LECTURE 9



Covers …

JDBC (small parts of Chapter 10)

Practical Guidance and More Detail for JDBC:
https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html

Much More on JPA (only touched upon in lecture):
https://www.tutorialspoint.com/jpa/
https://docs.oracle.com/javaee/7/tutorial/partpersist.htm#BNBPY

What we will be covering

How to interact with your database from Java using JDBC

Interfacing to your Database from Java

A database in isolation isn't very interesting

We usually want to **do something** with our database, which usually means interfacing to it from a larger application

Obviously this larger application can be written in any programming language, but we will be focusing on **Java** in this course

Using SQL in Programming

The fundamental interface between programming languages and DBMSs is, again, SQL

Three common alternatives:

- **Embedded SQL** (SQL as a **feature** of the host programming language)
- **Interpreted SQL** (SQL statements are **strings** in the host programming language, some library runs them against the DBMS)
- **“Hidden” SQL** (using some abstraction such as JPA to “hide” the SQL behind a nicer interface)
    - Usually maps to one of the above alternatives under the hood

Using SQL in Programming

The fundamental interface between programming languages and DBMSs is, again, SQL

Three common alternatives:

- **Embedded SQL** (SQL as a **fe** ing language)
- **Interpreted SQL** (SQL st atements are **st** mming language, some library runs them against the DBMS)
- **"Hidden" SQL** (using some abstraction such as JPA to "hide" the SQL behind a nicer interface)
  - Usually maps to one of the above alternatives under the hodd

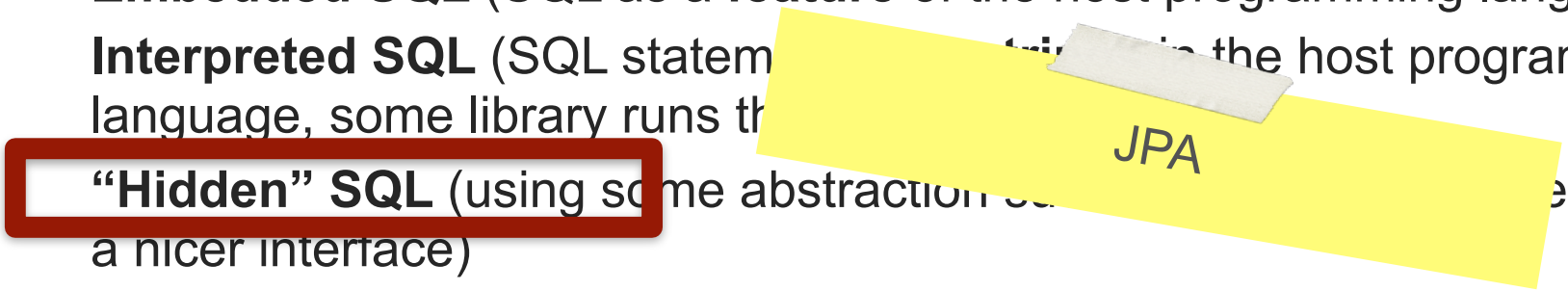
JDBC

Using SQL in Programming

The fundamental interface between programming languages and DBMSs is, again, SQL

Three common alternatives:

- **Embedded SQL** (SQL as a **feature** of the host programming language)
- **Interpreted SQL** (SQL statem… …the host programming language, some library runs th…
- **"Hidden" SQL** (using so…me abstraction … SQL behind a nicer interface)
  - Usually maps to one of the above alternatives under the hood



High-Level Sequence in Database Programming

1. Open a **connection** to database server
2. Interact with database by **submitting** queries, updates, and other database commands
    - In interpreted SQL, commands are given as strings
    - This is where everything from the last lectures comes in
3. Terminate or **close connection** to database

JDBC Example

```
public class PostgreSQLJDBC {
   public static void main( String args[] ) {
      Connection c = null;
      Statement stmt = null;
      try {
         Class.forName("org.postgresql.Driver");
         c = DriverManager
            .getConnection("jdbc:postgresql://localhost:5432/company",
            "philipp", "123");

         stmt = c.createStatement();
         ResultSet rs =
           stmt.executeQuery( "SELECT * FROM EMPLOYEE;" );
            while ( rs.next() ) {
                  String ssn = rs.getString("Ssn");
                  int salary = rs.getInt("Salary");
                 // do stuff with results
            }
            rs.close();
            stmt.close();
            c.close();
      } catch ( Exception e ) {
            System.err.println("Error "+e.getMessage());
            System.exit(0);
       }}}
```

JDBC Example

```
public class PostgreSQLJDBC {
  public static void main( String args[] ) {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.postgresql.Driver");
      c = DriverManager
        .getConnection("jdbc:postgresql://localhost:5432/company",
        "philipp", "123");

      stmt = c.createStatement();
      ResultSet rs =
        stmt.executeQuery( "SELECT * FROM EMPLOYEE;" );
      while ( rs.next() ) {
          String ssn = rs.getString("Ssn");
          int salary = rs.getInt("Salary");
          // do stuff with results
      }
      rs.close();
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println("Error "+e.getMessage());
      System.exit(0);
    }}}
```

Anatomy of a JDBC Interaction - Loading the Driver Class

Before doing anything with JDBC, you need to **load** the correct driver

```
Class.forName("org.postgresql.Driver");
```

Concrete fully qualified class name (FQN) depends on database driver (look up in documentation)

JAR file needs to be on classpath (no JDBC implementation is part of standard SDK)

JDBC Example

```
public class PostgreSQLJDBC {
  public static void main( String args[] ) {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.postgresql.Driver");
      c = DriverManager
        .getConnection("jdbc:postgresql://localhost:5432/company",
        "philipp", "123");

      stmt = c.createStatement();
      ResultSet rs =
        stmt.executeQuery( "SELECT * FROM EMPLOYEE;" );
      while ( rs.next() ) {
          String ssn = rs.getString("Ssn");
          int salary = rs.getInt("Salary");
          // do stuff with results
      }
      rs.close();
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println("Error "+e.getMessage());
      System.exit(0);
    }}}
```

Anatomy of a JDBC Interaction - Getting a Connection with JNDI

DriverManager.getConnection(<JNI-String>,<user>,<password>)

To open a connection use the static method `getConnection` from the `DriverManager` class.

- This uses the implementation that was loaded before with `forName`
- If you want to have different JDBC drivers at the same time you need multiple classloaders (out of scope in this course)

Anatomy of a JDBC Interaction - Getting a Connection with JNDI

The first parameter is a **JNDI identification string** that allows Java to figure out what database instance to connect to

- You **can** actually connect to multiple databases at the same time very easily, as long as they all work with the same driver

Details of JNDI (**Java Naming and Directory Interface**) are not important here, but it's a standardized way to look up things in Java

- Concrete format differs from JDBC driver to driver
- For PQSQL driver:

```
jdbc:postgresql://<HOST>:<PORT>/<DATABASE>
```

JDBC Example

```
public class PostgreSQLJDBC {
   public static void main( String args[] ) {
      Connection c = null;
      Statement stmt = null;
      try {
         Class.forName("org.postgresql.Driver");
         c = DriverManager
            .getConnection("jdbc:postgresql://localhost:5432/company",
            "philipp", "123");

      stmt = c.createStatement();
      ResultSet rs =
        stmt.executeQuery( "SELECT * FROM EMPLOYEE;" );
            while ( rs.next() ) {
                  String ssn = rs.getString("Ssn");
                  int salary = rs.getInt("Salary");
                 // do stuff with results
            }
            rs.close();
            stmt.close();
            c.close();
      } catch ( Exception e ) {
            System.err.println("Error "+e.getMessage());
            System.exit(0);
       }}}
```

Anatomy of a JDBC Interaction - Running a Query

Two-step procedure:

1. Create a **statement** object
2. **Execute** the statement with a SQL query given as String

```
stmt = c.createStatement();
ResultSet rs =
   stmt.executeQuery( "SELECT * FROM EMPLOYEE;" );
```

Anatomy of a JDBC Interaction - Running a Query

Statement can be any kind of (legal) SQL code:

`SELECT, INSERT INTO, UPDATE, DELETE, …`
(less common: `CREATE TABLE, CREATE VIEW`, etc.)

Two different methods used to invoke SQL queries and other commands:

```
ResultSet rs = stmt.executeQuery(<QUERY>);
int changedItems = stmt.executeUpdate(<SQL>);
```

ResultSets and Cursors

Queries return a **ResultSet**

- Basically a linked list of rows

You iterate over them using a **cursor**

- The cursor tells you which row you are currently looking at
- Cursor typically only allows you to **move forward**

```
rs.next()
```

Impedance Mismatch

There are **differences** between database model and programming language model

- Relations vs. objects

Binding for each host programming language

- Specifies for each attribute type the compatible programming language types

SQL INTEGER → java.lang.Integer
SQL DATE → java.util.Date
SQL BINARY → java.lang.Byte[]

JDBC Example

```
public class PostgreSQLJDBC {
  public static void main( String args[] ) {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.postgresql.Driver");
      c = DriverManager
        .getConnection("jdbc:postgresql://localhost:5432/company",
        "philipp", "123");

      stmt = c.createStatement();
      ResultSet rs =
        stmt.executeQuery( "SELECT * FROM EMPLOYEE;" );
      while ( rs.next() ) {
          String ssn = rs.getString("Ssn");
          int salary = rs.getInt("Salary");
          // do stuff with results
      }
      rs.close();
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println("Error "+e.getMessage());
      System.exit(0);
    }}}
```

Anatomy of a JDBC Interaction - Closing the Connection

After **the last** JDBC statement the connection should be closed

```
c.close();
```

One connection can be **reused** for multiple statements

- But typically not at the same time

Opening and closing connections is **expensive**

- Use connection pooling

Transactions in JDBC

More on the principles behind transactions next lecture

Default mode is **auto-commit**

All your statements are run in their own transaction

However, can also be **manually managed**

Transactions in JDBC

```
try {
        conn.setAutoCommit(false);
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(SQL);
        stmt.executeUpdate(SQL2);
        conn.commit();
} catch (Exception e) {
         conn.rollback():
}
```

Note the '?' in the statement string.

Prepared Statements

Often we need to compose a query a program runtime:

E.g., find the employee with a last name as stored in a variable

Rather than assembling a query string, it is better to use a **prepared statement** with placeholders:

```
String name = "Leitner";
PreparedStatement stmt = conn.prepareStatement(
        "SELECT * FROM EMPLOYEE WHERE Lname = ?");
stmt.setString(1, name);
ResultSet rs = stmt.executeQuery();
```

Advantages of Prepared Statements

(1) Less cluttered, communicates intend better

(2) Faster (gets precompiled and cached at database level)

(3) Avoids many types of SQL Injection attacks

Relationship between Java and Database Concepts

Typically, there is an **implicit mapping** between Java and database concepts:

Java classes → database tables (class Employee → EMPLOYEE)

Object instances → table rows (Employee emp → EMPLOYEE[1])

Class fields → table attributes (emp.lname → EMPLOYEE.Lname)

Object relationships → associations

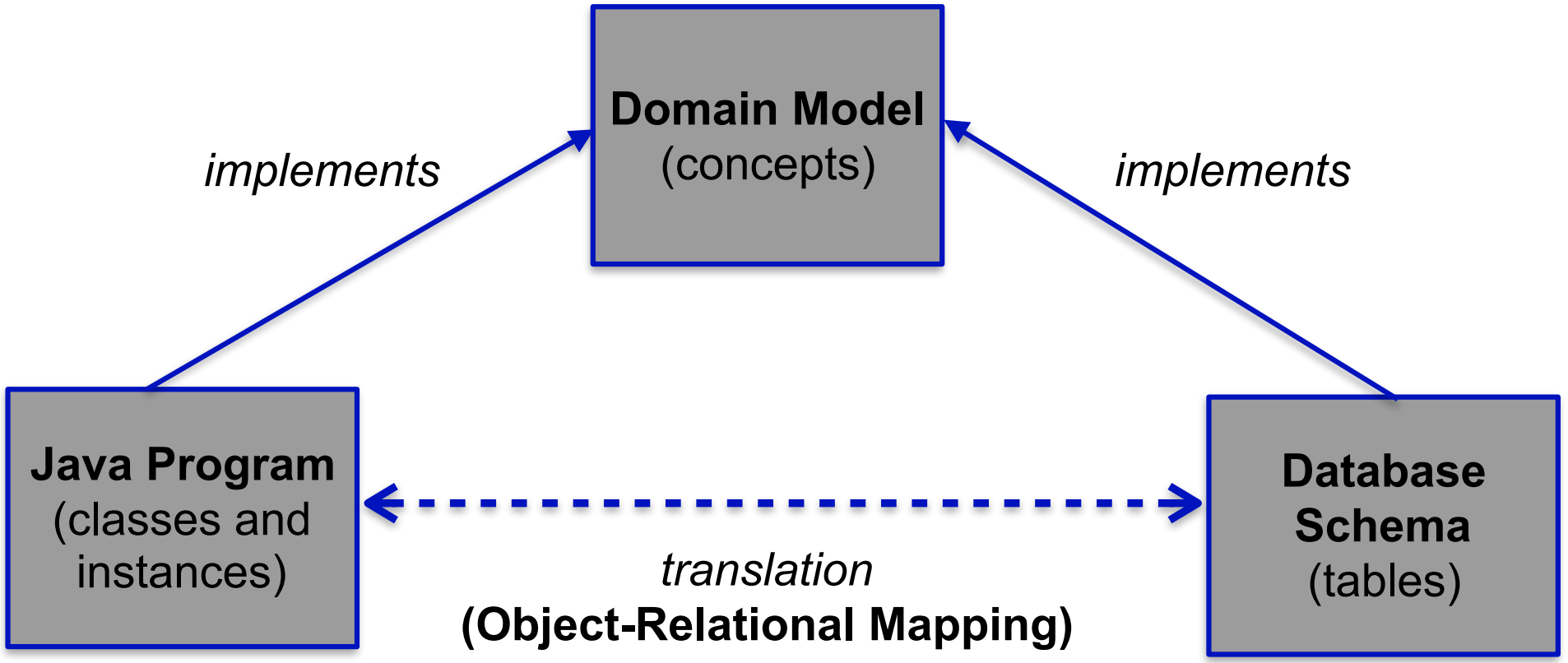
(Employee emp.supervisor → PK/FK relationship)

Relationship between Java and Database Concepts

On “diagram level” this relationship is usually easy to see:

EER diagram of database and UML class diagram of Java program look (almost) the same

Think of Java program and database schema as **two technology mappings** (or implementations) of the same domain concepts


Domain Model
(concepts)
implements
implements
Java Program
(classes and
instances)
Database
Schema
(tables)
translation
(Object-Relational Mapping)

Object-Relational Mapping (ORM) using JDBC

Using JDBC, implementing the object-relational mapping is the task of the developer

```
ResultSet rs = … // some query
List<Employee> results = new LinkedList<Employee>();
while ( rs.next() ) {
        Employee emp = new Employee();
        emp.setSsn(rs.getString(“Ssn”));
        emp.setSalary(rs.getInt(“Salary"));
        // and so on
        results.add(emp);
}
```

Java Persistence API (JPA)

JPA is a **Java specification** (not part of standard library!) which does this automatically

Declarative instead of imperative

Magic SQL Strings partially replaced with native Java constructs

However, queries still end up as Strings

Java Persistence API (JPA)

Very small example of a JPA **entity class**

```
@Entity
class Employee {
    @Id  private String ssn;
    @NotNull private String lname;
     private String fname;

     // getters and setters for all fields
}
```

generates

```
CREATE TABLE Employee (
    ssn VARCHAR(30) PRIMARY KEY,
    lname VARCHAR(30) NOT NULL,
    fname VARCHAR(30
);
```

Java Persistence API (JPA)

Using this entity class in a program

```
EntityManager em = … // obtain an entity manager
Employee emp = em.find(Employee.class, “56834678”);
emp.setFname(“Philipp Wolfgang”);
em.merge(emp);
```

The annotations and EntityManager classes are standardized in the JSR, but how to actual SQL is generated is (largely) up to the implementation.

Java Persistence API (JPA)

Note that JPA "knows" from the mapping that the primary key of Employee is ssn - no need to tell.

```
EntityManager em = … // obtain an entity manager
Employee emp = em.find(Employee.class, "56834678");
emp.setFname("Philipp Wolfgang");
em.merge(emp);
```

generates

```
SELECT * FROM Employee
WHERE ssn = "56834678";
```

Java Persistence API (JPA)

```
EntityManager em = … // obtain an entity manager
Employee emp = em.find(Employee.class, “56834678”);
emp.setFname(“Philipp Wolfgang”);
em.merge(emp);
```

generates

```
UPDATE Employee
SET ssn = “56834678”,
  lname = “Leitner”,
  fname = “Philipp Wolfgang”
WHERE ssn = “56834678”;
```

Mapping Relationships in JPA

```
@Entity
class Employee {
    @Id  private String ssn;
    // … other attributes

    @ManyToOne
    Employee supervisor;

    @ManyToMany
    List<Department> departments;
}
```

This is the most simple case, there are many more details that often need to be specified in practice

Law of Leaky Abstractions

This is basically the **"Law of Leaky Abstractions"** by Joel Spolsky

https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

Essentially, every non-trivial abstraction in computer science requires users to **understand the technology they are hiding**.

Otherwise, **bugs and performance problems** tend to creep up.

Key Takeaways

JDBC is the **low-level interface** to execute arbitrary SQL statements from Java

- You don’t need to remember the API by heart, but you should know the basic steps and principles
- In most cases you should use **prepared statements**

There is an **impedance mismatch** to resolve between SQL and Java programs

JPA is an attempt to resolve this impedance mismatch. Most industrial database project use it (or a variation of the idea)

- However, you can’t use JPA productively **without intimate understanding** of SQL