Data persistence

Low-level access to RDBMS –

Technology case study: JDBC

Outline

- The problem of Data Persistence
- Short Review of RDBMS
- JDBC (Java DataBase Connectivity)

What is “Data persistence”

- Programs (Object-oriented programs) manipulate **data** (represented as **object state**) that need to be **stored:**
  - **persistently** - to survive when app gets stopped
  - **queriably** - to be able to retrieve/access them
  - **scalably** - to be able to handle large data volumes
  - **transactionally** - to ensure their consistency

- Different types of storage can be used (files, databases)

Types of Storage

- Serialization:
  - simple, yet hardly queriable, not transactional, ...
  - stream persisting an instance of class C is deprecated once definition of C is modified (e.g. field added/removed).
- Files of standardized formats: xml, json
  - Readable format; verbose, inefficient storage for large data
- Relational Databases (MySQL, PostgreSQL, Oracle, ...)
  - efficient storage for data with rigid schema
  - efficient search using SQL standard
  - secure and transactional
- NoSQL Databases
  - Key-value storages (MongoDB, Hadoop, ...)
    - suitable for data without rigid schema
  - Object Databases
    - designed in 90's to capture complexity of object models
    - Issues: scalability, standardized queries

Issues related to Data persistence

- **Methods/Technologies for physically accessing external storage:**
  - **Low-level APIs providing programmatic access to external storage** from within OO programs:
    - Files – parsers
    - Databases – ODBC, JDBC

- **Design Problems for OO applications which use data persistency:**
  - **Decoupling -** separate business logic from storage
  - **Abstraction** – hide storage details
  - **Mapping** between OO domain model and the model used by a non-OO persistency technology (OO-XML, OO-relationalDB)

Persistence Challenges when using Relational Databases

- How to connect to a database from within the OO program?
- How to send queries/ how to receive results?
- How to make sure that the queries are correct?
- How to map objects to relations?
- How to abstract the application from the underlying database?

Accessing Relational Databases

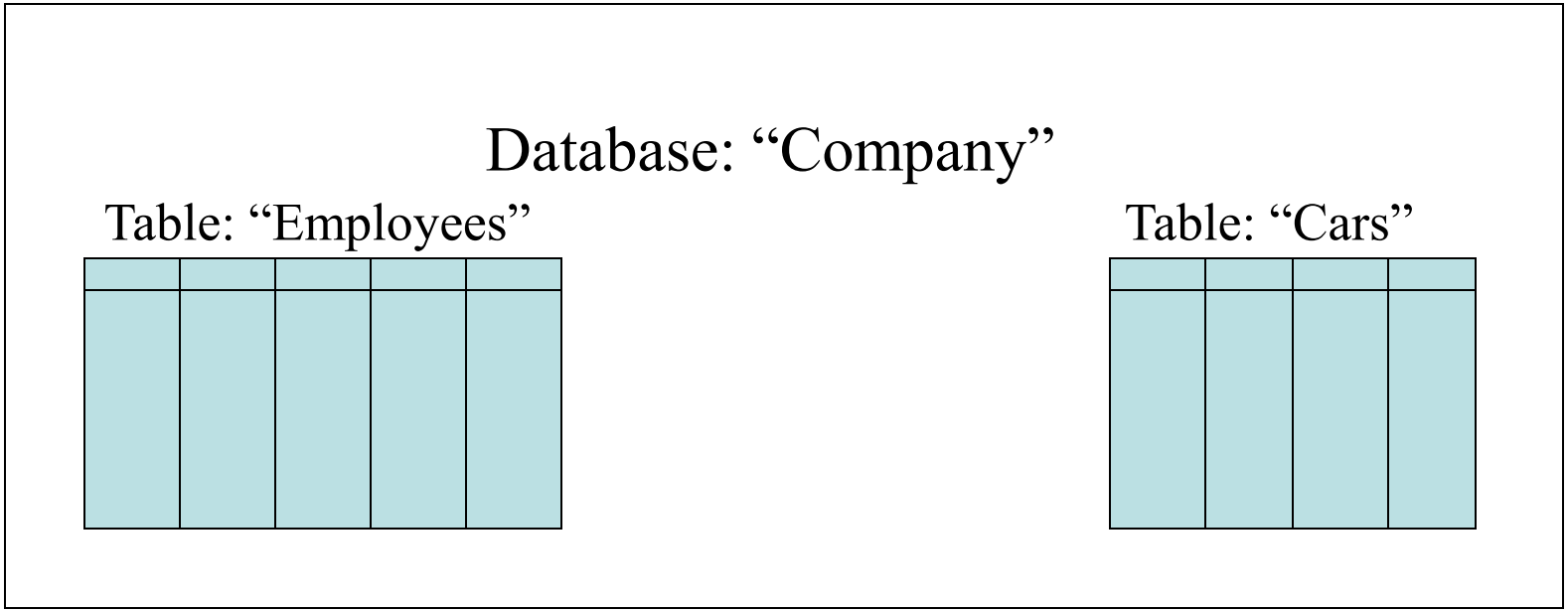
- Need to **standardize the ways OO applications access relational databases:**
  - Low-level standard: ODBC (Open DataBase Connectivity)
    - A low-level API for managing connection, authorization, query and result delivery introduced by Microsoft in early 1990s
  - Language-specific low-level APIs:
    - connect to databases, run SQL queries, and read results. Often use native drivers as backend to connect to database.
    - In java: JDBC
    - .NET: ADO.NET
    - PHP: PDO
    - JavaScript: Node.js
- **Higher level of abstraction**:
  - Map directly objects <-> records (Object-Relational Mappers - ORMs)
  - Abstract data source (DAO, Repository pattern)

Very Short Review of Relational Databases

Relational Databases

- *Relational Database*s:
  - Information is organized in *Table*s
  - A Table has a name and several *Columns*
  - Each Column has a name and a data type
  - Each data is stored on a Row in a table
  - A table = a relation = a collection of objects of the same type on rows

Example



Example

Table Employees

Employee_Number	First_name	Last_Name	Date_of_Birth	Car_Number
10001	Axel	Washington	28-Aug-43	5
10083	Arvid	Sharma	24-Nov-54	null
10120	Jonas	Ginsberg	01-Jan-69	null
10005	Florence	Wojokowski	04-Jul-71	12

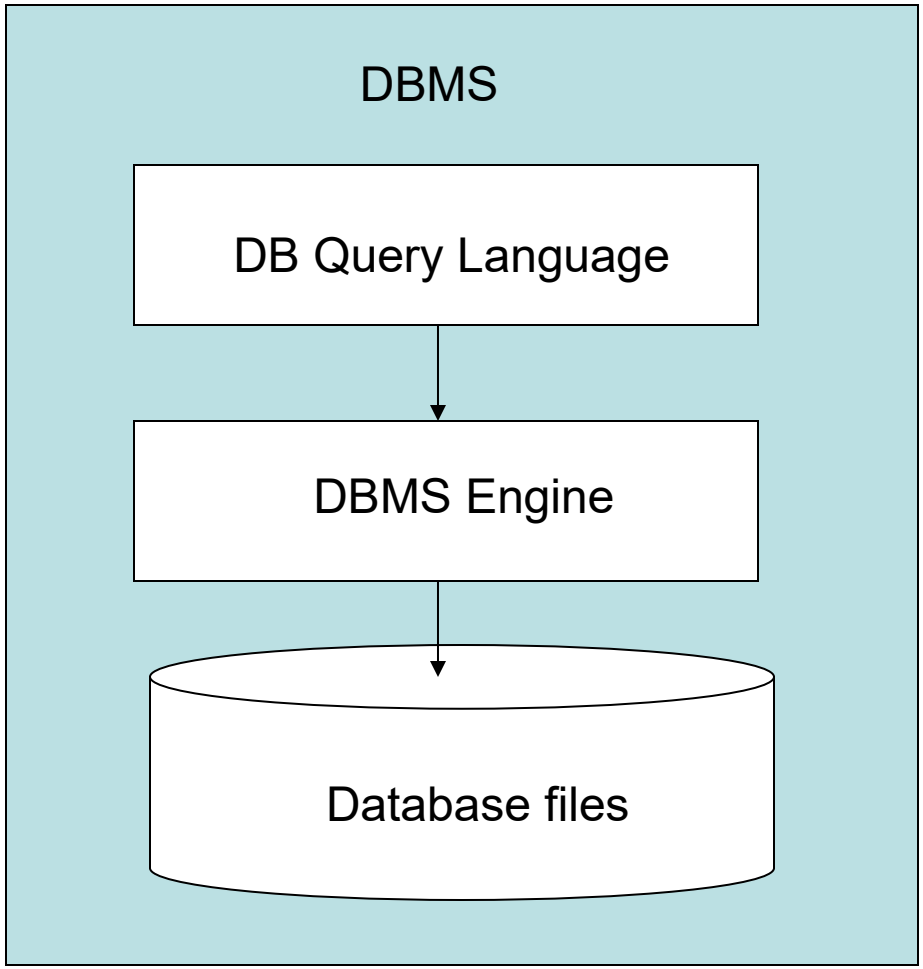
Table Cars

Car Number	Make	Model	Year
5	Honda	Civic DX	1996
12	Toyota	Corolla	1999

Integrity rules

- A table cannot have two identical rows (DBMS verifies this)
- Values in a column cannot be arrays
- Null values: a special value to show a missing element (different from zero or blanc !)
- *Primary key*: one or several columns that can be used to identify (differentiate) elements (rows). The columns that are part of the primary key cannot have null values
- *Foreign key*: A foreign key is a column that references the primary key of another table.
  - Example: Car_Number: primary key in table Cars, foreign key in table Employees

DataBase Management Systems



SQL: general presentation

- SQL (Structured Query Language): a standard language to interrogate databases
- SQL commands - categories:
  - **Data manipulation –** add, retrieve, modify or delete data; these are commands for database users.
    - `SELECT, UPDATE, DELETE, INSERT`
  - **Data definition –** create, configure or delete tables, databases, views, or indexes; these commands manage the database
    - `CREATE, DROP`

SQL: SELECT

- `SELECT` is the command used for queries; it selects values from the specified columns, for all rows that match the condition given by the WHERE clause

- Example:

```
SELECT First_Name, Last_Name FROM Employees WHERE
    Car_Number IS NOT NULL
SELECT * FROM Employees
```

SQL: The clause WHERE

- The clause WHERE describes conditions
- This clause can be used together with the commands SELECT, UPDATE, DELETE

- Example:

```
SELECT First_Name, Last_Name FROM Employees WHERE
    Last_Name LIKE 'Washington%'
SELECT First_Name, Last_Name FROM Employees WHERE
    Last_Name LIKE ‘Ba_man‘
SELECT First_Name, Last_Name FROM Employees WHERE
    Employee_Number < 10100 and Car_Number IS NULL
```

SQL: Join

- Retrieves data from more than one table in one query
- Example:
- Finds employees that have cars and shows: name of employee (from table employees), number of car, brand, model and manufacturing year (from table cars)
- ```
  SELECT Employees.First_Name, Employees.Last_Name,
  Cars.Make, Cars.Model, Cars.Year FROM Employees,
  Cars WHERE Employees.Car_Number = Cars.Car_Number
  ```
  ```

SQL: INSERT

- Command INSERT is used for adding new rows in tables

- Example

```
insert into SUPPLIERS values(49, 'Superior Coffee',
  '1 Party Place', 'Mendocino', 'CA', '95460');

insert into COFFEES values('Colombian', 00101, 7.99,
  0, 0);
```

SQL: UPDATE

- Command UPDATE is used to modify values in tables

- Example:

```
UPDATE suppliers
  SET zip = '99999'
  WHERE zip = '95460'
UPDATE coffees SET price = 9.99
  WHERE price<=5.0
```

SQL: DELETE

- Command DELETE deletes rows from tables

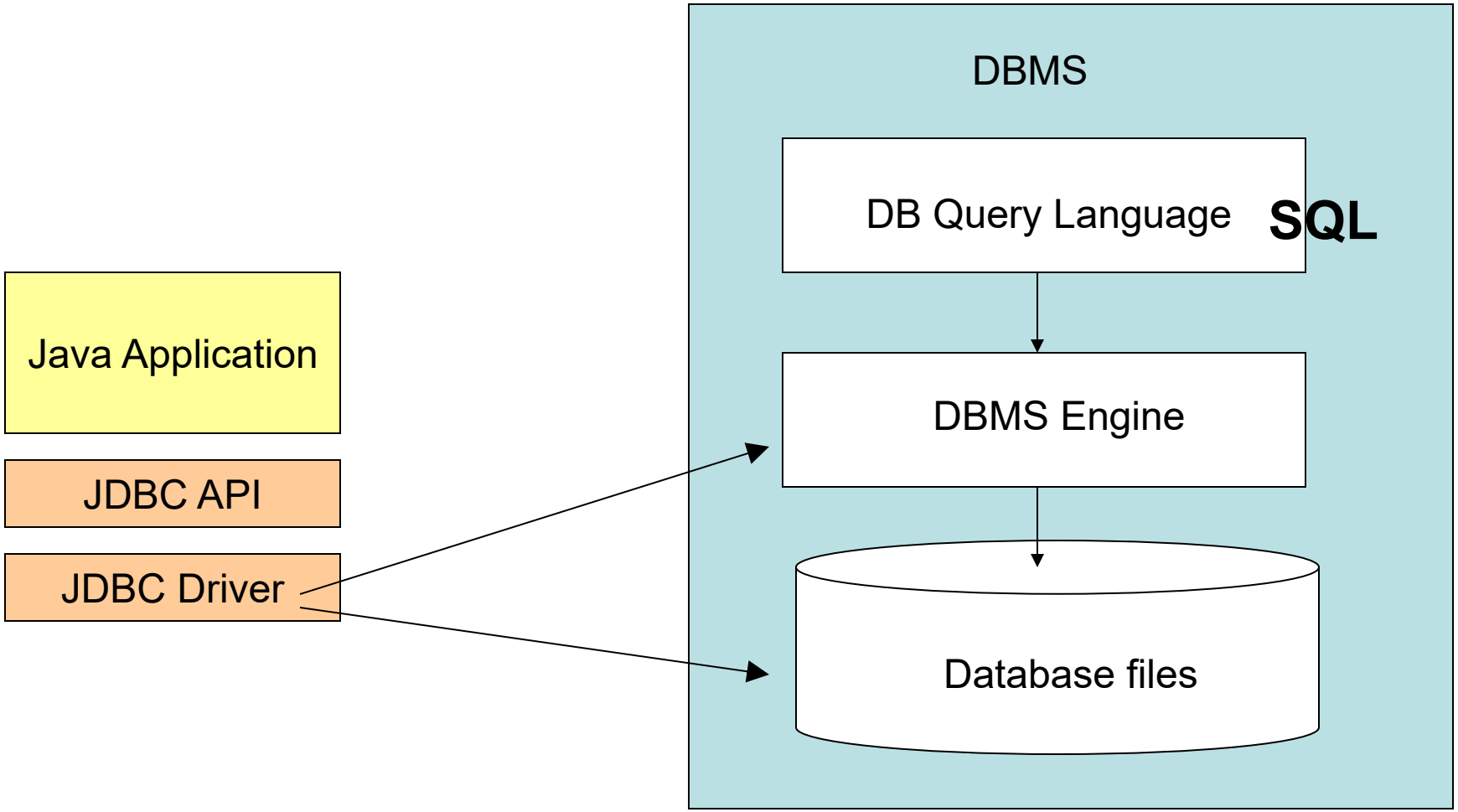
```
DELETE FROM coffees  WHERE price > 100
```

SQL: CREATE

```
create table SUPPLIERS (
   SUP_ID integer NOT NULL, SUP_NAME varchar(40)
   NOT NULL, STREET varchar(40) NOT NULL, CITY
   varchar(20) NOT NULL, STATE char(2) NOT NULL,
   ZIP char(5), PRIMARY KEY (SUP_ID));

create table COFFEES (
   COF_NAME varchar(32) NOT NULL, SUP_ID int NOT
   NULL, PRICE numeric(10,2) NOT NULL, SALES
   integer NOT NULL, TOTAL integer NOT NULL,
   PRIMARY KEY (COF_NAME), FOREIGN KEY (SUP_ID)
   REFERENCES SUPPLIERS (SUP_ID));
```

JDBC:
Java <-> Relational Databases



JDBC

JDBC

Bibliography:

Java tutorial - trail on JDBC:
http://download.oracle.com/javase/tutorial/jdbc/index.html

H2 Database engine:
https://www.h2database.com/html/main.html

Apache Derby tutorial:
http://db.apache.org/derby/papers/DerbyTut/index.html

What is JDBC

- JDBC is a Java **API** that can access any kind of tabular data, especially data stored in a Relational Database
- JDBC helps you to write Java applications that can:
  - Connect to a data source, like a database
  - Send queries and update statements to the database
  - Retrieve and process the results received from the database in answer to your query

- JDBC is **just the API.** You also need:
  - The Relational Database Management System
  - A matching JDBC Driver (the implementation of the API)

Which Database?

- **Production DBMS (server-based)**
  - Examples: MySQL, Oracle
  - Run as **database servers**
  - Handle remote access, concurrency, security
- **Lightweight / Embedded databases**
  - Examples: SQLite, H2, Apache Derby
  - Can run embedded (inside application), no need for database server, application accesses directly the database files
  - Some (H2, Derby) can also run in server mode
- **In-memory databases**
  - Example: H2
  - Data stored in **RAM only** - Not persistent !
  - Used for learning, testing and prototyping

API vs Implementation

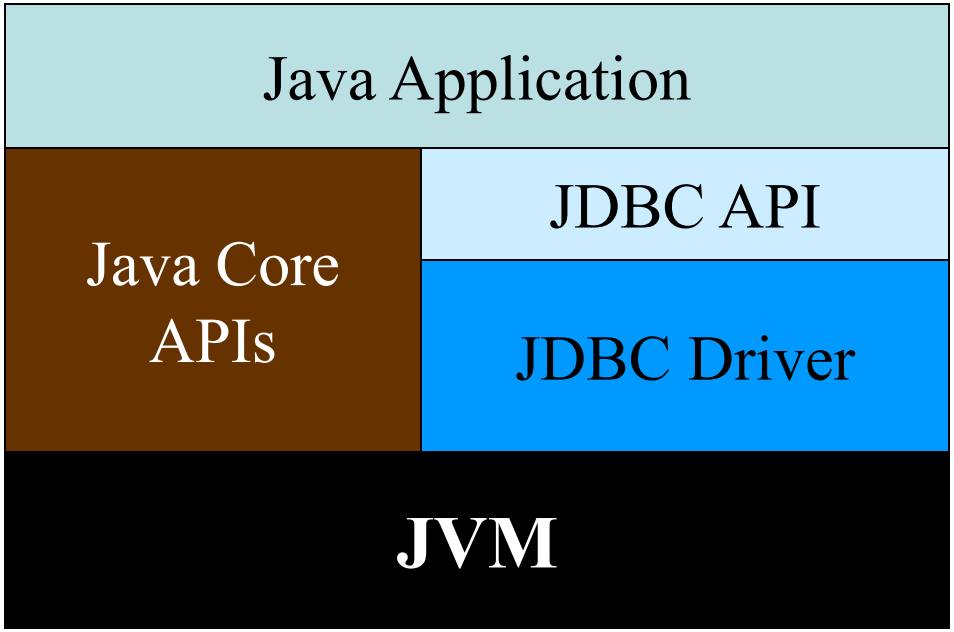
- Many Java technologies are defined as APIs:
  - Examples: JDBC, JPA, JAXP, …
- API vs Implementation: ensures decoupling and flexibility: one can use different third party implementations
- Bridge Pattern: a structural design pattern used to decouple an abstraction from its implementation
- Factory Pattern: used to instantiate the right implementations of the abstractions in the API

JDBC Drivers

- The JDBC **API:** the java.sql package:
  - contains several **interfaces**, (such as Connection, Statement, and ResultSet)
  - defines an abstract API that requires a concrete **implementation**, which is provided by the JDBC driver.
- A **JDBC driver:**
  - is a software component that allows Java applications to connect to a *specific* relational database
    - Each RDMS needs its own JDBC driver. The JDBC driver is usually provided by the database producer.
  - acts as a **bridge** between the application and the database.
  - comes in the form of a **jar** that must be included in the project's classpath (or set as a mvn dependency)

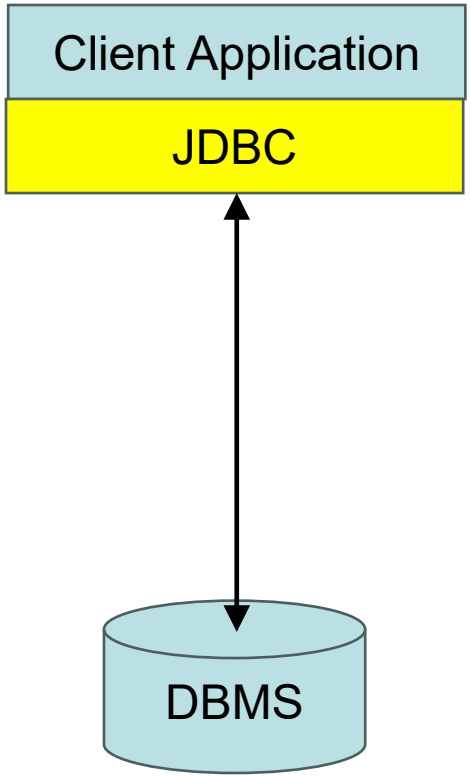
The JDBC Driver

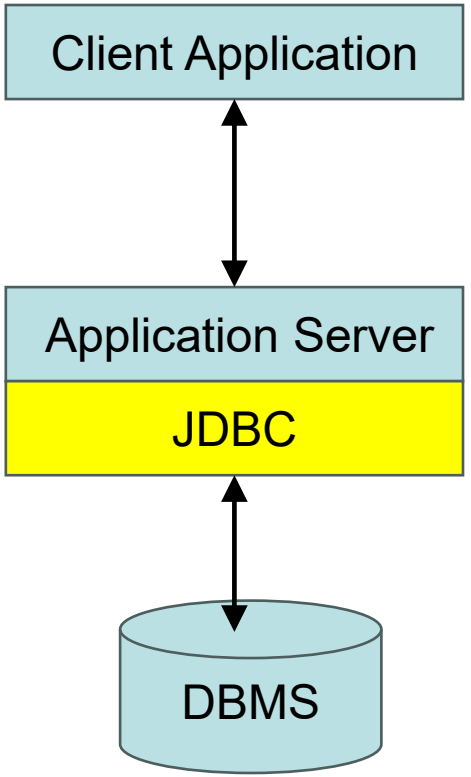
- JDBC Driver: facilitates the communication with a certain type of database server
- JDBC Driver can be realized in several ways



JDBC Architecture

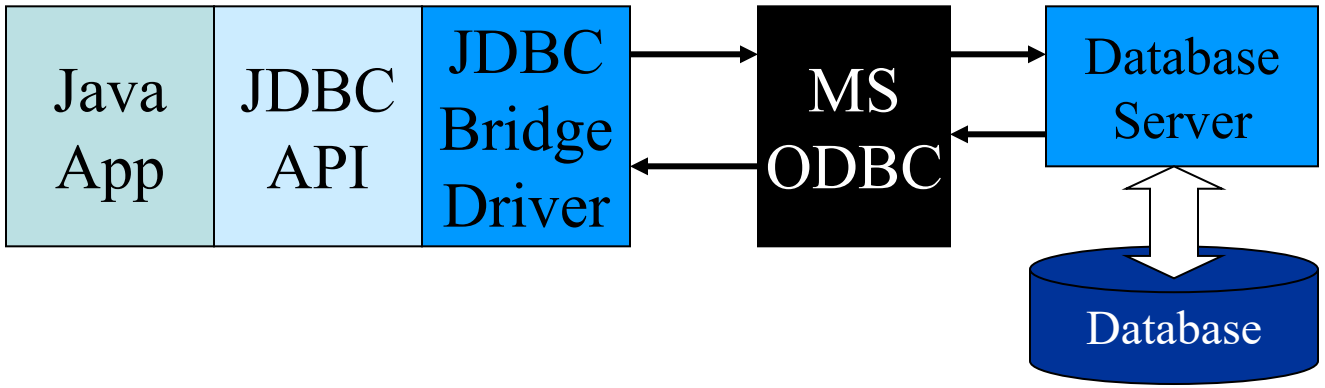
- The JDBC API supports both **two-tier** and **three-tier** processing models for database access





Types of JDBC Drivers - (1)

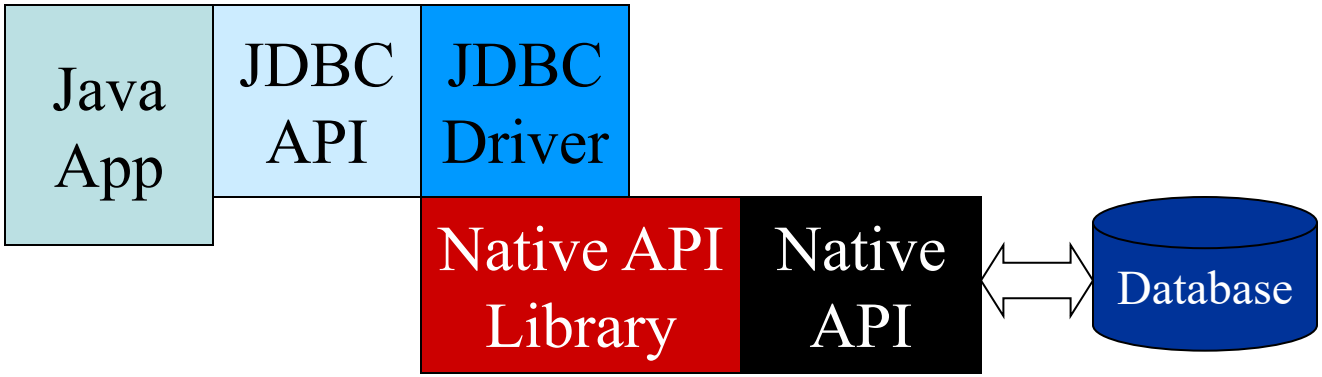
Type 1:
ODBC
Bridge



Type 1 JDBC drivers: ODBC Bridge: Drivers that implement the JDBC API as a mapping to another data access API, such as ODBC (Open Database Connectivity). Drivers of this type are generally dependent on a native library, which limits their portability. The JDBC-ODBC Bridge is an example of a Type 1 driver.

Types of JDBC Drivers - (2)

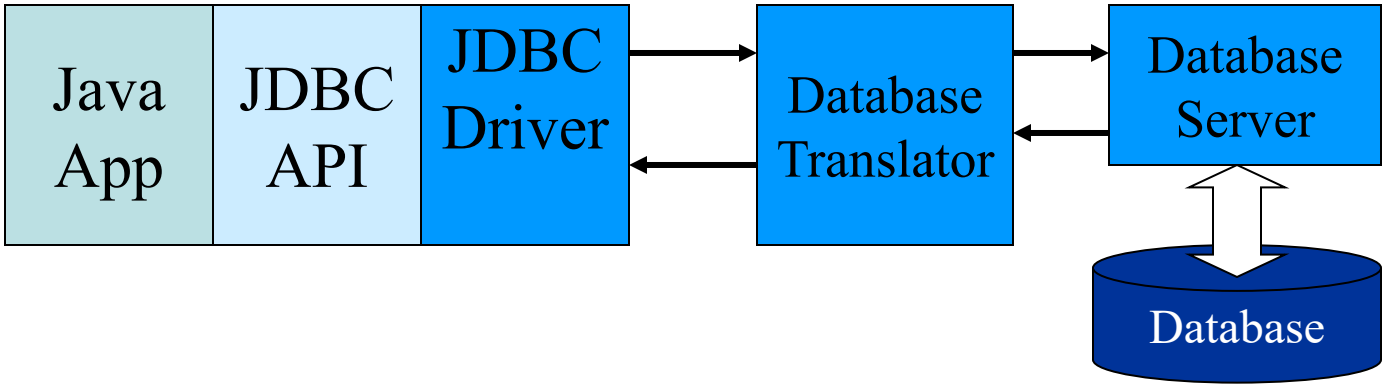
Type 2:
Native
DB API



Type 2 JDBC drivers: Native DB API: Drivers that are written partly in the Java programming language and partly in native code. These drivers use a native client library specific to the data source to which they connect. Again, because of the native code, their portability is limited. Oracle's OCI (Oracle Call Interface) client-side driver is an example of a Type 2 driver.

Types of JDBC Drivers - (3)

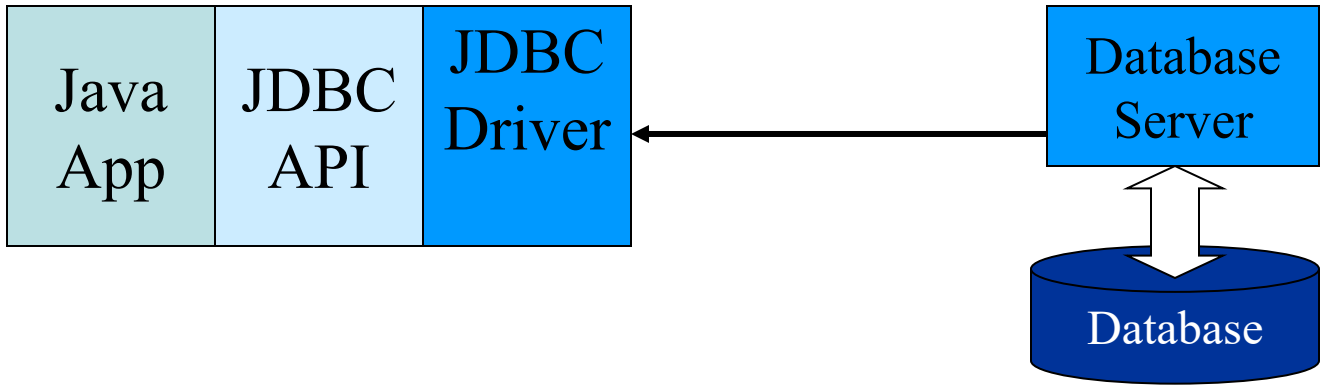
Type 3:
Pure
Java



Type 3 JDBC drivers: Pure Java: Drivers that use a pure Java client and communicate with a middleware server using a database-independent protocol. The middleware server then communicates the client's requests to the data source.

Types of JDBC Drivers - (4)

Type 4:
Pure
Java
Direct



Type 4 JDBC drivers: Pure Java Direct: Drivers that are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source.

Java DB (Derby) comes with two Type 4 drivers, an Embedded driver and a Network Client Driver. MySQL Connector/J is a Type 4 driver.

A typical JDBC client program

- The typical sequence of operations when interacting with a database via JDBC:
  - **Load the JDBC driver** that matches with the database system
  - Open a ***Connection*** to the database (using URL)
  - Creates a ***Statement*****,** using the Connection object obtained in the previous step
  - *Execute* the statement
  - Process the results which have been returned through a ***ResultSet***
  - Close Connection

Connecting to the datasource

- A data source can be: a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver.
- Load the JDBC Driver:
  - Ensures Java knows how to communicate with your database. For JDBC 4.0+, drivers are auto-loaded if present in the classpath.
- Establish a Connection
  - Use `DriverManager.getConnection`
  - This method requires a ***ConnectionURL***, which varies depending on your DBMS
  - A **JDBC ConnectionURL** is a string that tells the driver:
    - Where the database is located
    - Which database to use
    - Optional configuration settings (user, password, other connection properties)

Specifying Connection URL's

- The exact syntax(format) of a database connection URL is specified by your DBMS !

- MySQL Connector/J Database URL format:
  - jdbc:mysql://[*host*][,*failoverhost*...][:*port*]/[*database*][?*propertyName1*][=*propertyValue1*][&*propertyName2*][=*propertyValue2*]...
  - Example: jdbc:mysql://localhost:3306/

- H2
  - jdbc:h2:[mem|file:][databaseName][;options]
  - Example – in memory DB: jdbc:h2:mem:testdb
  - Example – in file DB: jdbc:h2:file:C:/Users/ioana/test

- Apache Derby Connection URLs format:
  - jdbc:derby:[*subsubprotocol*:][*databaseName*][;*attribute*=*value*]*
  - Example : jdbc:derby:*testdb*;create=true

Establishing Connection – Example with MySQL

```
Connection conn = null;

Properties connectionProps = new Properties();
connectionProps.put("user", userName);
connectionProps.put("password", password);

conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/",
    connectionProps
);
```

DriverManager selects a registered JDBC driver based on the JDBC URL.
Then it delegates the creation of the Connection object to that driver. This is classic **Factory Method** pattern: the client asks a central factory (DriverManager) for an object (Connection), without knowing which concrete implementation will be used.

The JDBC Connection Class

- There are many methods a program can call on its valid Connection object.
  - The **createStatement()** method will create a Statement object that can be used to assemble and run SQL commands. The **preparedStatement()** creates an object that is associated with a predefined SQL command (the Statement object can be used for arbitrary statements – and can be reused for other SQL commands)
  - The **getMetaData()** method will return metadata associated with the database, including descriptions of all of the tables in the DB.
  - The **prepareCall()** method is used to call stored procedures in the SQL database.

Creating Statements

- A Statement is an interface that represents a SQL statement.
- You execute Statement objects, and they generate ResultSet objects, which is a table of data representing a database result set.
- You need a Connection object to create a Statement object.

- **Example:**

```
Connection con;
Statement stmt;
…
con = DriverManager.getConnection( … );
…
stmt=con.createStatement();
```

Executing Statements/Queries

- To execute a query, call an execute method from Statement such as the following:
  - **executeQuery**: Returns one ResultSet object. Use this method if you are using SELECT
  - **executeUpdate**: Returns an integer representing the number of rows affected by the SQL statement. Use this method if you are using INSERT, DELETE, or UPDATE SQL statements.

- ResultSet: A ResultSet is like a database table; it has zero or more rows (zero if no data elements match the query criteria) and each row has one or more columns

- Example:

```
String query = "select COF_NAME, SUP_ID, PRICE, SALES,
  TOTAL from COFFEES";
ResultSet rs = stmt.executeQuery(query);
```

Processing ResultSets

- You access the data in a **ResultSet** object through a **cursor**. This cursor is a pointer that points to one row of data in the ResultSet object.
- Initially, the cursor is positioned before the first row. You call various methods defined in the ResultSet object to move the cursor.
- For example, the method **ResultSet.next** moves the cursor forward by one row
- There are multiple methods of extracting data from the current row in a ResultSet.
  - The **getString()** method returns the value of a particular column in the current row as a String.
  - The **getInt()** method returns the value of a particular column in the current row as an int.
  - The **getBoolean()** method returns the value of a particular column in the current row as an boolean.
  - The **getDouble()** method returns the value of a particular column in the current row as an double.
  - The **getObject()** method returns the value of a particular column in the current row as an Object.

ResultSet Example

```
String query = "select COF_NAME, SUP_ID, PRICE, SALES,
TOTAL from COFFEES";

stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
       String coffeeName = rs.getString("COF_NAME");
       int supplierID = rs.getInt("SUP_ID");
       float price = rs.getFloat("PRICE");
       int sales = rs.getInt("SALES");
       int total = rs.getInt("TOTAL");
       System.out.println(coffeeName + "\t" + supplierID +
                    "\t" + price + "\t" + sales + "\t" +
total);
```

Populating Tables Example

```
stmt = con.createStatement();
stmt.executeUpdate("insert into COFFEES” +
      "values('Colombian', 00101, 7.99, 0, 0)");
stmt.executeUpdate("insert into COFFEES“ +
      “values('French_Roast', 00049, 8.99, 0, 0)");
```

Creating Tables Example

```
String createString = "create table " + dbName + ".COFFEES " +
"(COF_NAME varchar(32) NOT NULL, " +
"SUP_ID int NOT NULL, " +
 "PRICE float NOT NULL, " +
"SALES integer NOT NULL, " +
"TOTAL integer NOT NULL, " +
"PRIMARY KEY (COF_NAME), " +
"FOREIGN KEY (SUP_ID) REFERENCES " + dbName +
".SUPPLIERS (SUP_ID))";

Statement stmt = con.createStatement();
stmt.executeUpdate(createString);
```

JDBC Statements - Types

- There are three types of statement objects in JDBC programming:
  - **Statement -** This allows the execution of simple SQL statements with no parameters.
  - **PreparedStatement -** (Extends Statement) Used for precompiling SQL statements that might contain input parameters. Allows the program to execute the same SQL command repeatedly, while allowing substitutions for particular words or values (input parameters).
  - **CallableStatement -** (Extends PreparedStatement.) Used to execute *stored procedures* that may contain both input and output parameters. A stored procedure is a function that is part of, and stored inside, a SQL database

Prepared Statement Example

```
PreparedStatement updateSales = null;

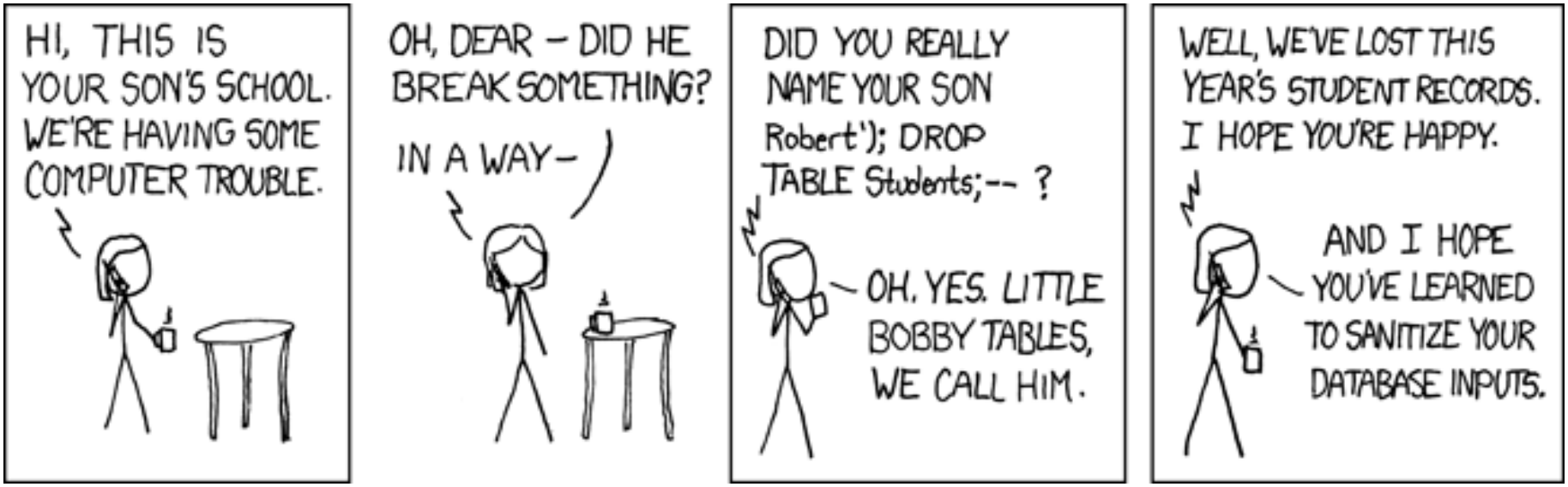
String updateString = "update " + dbName + ".COFFEES " +
                      "set SALES = ? where COF_NAME = ?";

con.setAutoCommit(false);

updateSales = con.prepareStatement(updateString);

for (Map.Entry<String, Integer> e : salesForWeek.entrySet())
   { updateSales.setInt(1, e.getValue().intValue());
      updateSales.setString(2, e.getKey());
     updateSales.executeUpdate();
    con.commit(); }
```

SQL Injection



SQL Injection

```
Scanner scanner = new Scanner(System.in);
String username = scanner.nextLine();
String password = scanner.nextLine();

String sql = "SELECT * FROM users WHERE username = '" +
        username + "' AND password = '" + password + "'";

System.out.println("Executing SQL: "+sql);

ResultSet rs = stmt.executeQuery(sql);

if (rs.next()) {
        System.out.println("Login successful!");
} else {
        System.out.println("Login failed.");
}
```

Dangerous: Query constructed by String concatenation without "sanitizing" user input

Input Username: `admin' OR '1'='1`
Input Password: `whatever`

```
Executing: SELECT * FROM users WHERE username = 'admin' OR '1'='1'
AND password = 'whatever'
Login successful!
```

Preventing SQL Injection with PreparedStatement

```
String sql = "SELECT * FROM users WHERE username = ? AND password =
?";

PreparedStatement ps = conn.prepareStatement(sql);

// bind values safely
ps.setString(1, username);
ps.setString(2, password);

System.out.println("Executing PreparedStatement: "+sql);

ResultSet rs = ps.executeQuery();

if (rs.next()) {
        System.out.println("Login successful!");
} else {
        System.out.println("Login failed.");
}
```

JDBC - Summary

- JDBC is a Java API that can access any kind of tabular data, especially data stored in a Relational Database
- In order to use JDBC with a particular database, you must install a driver for the database server you are using
- Makes interaction with database possible
- JDBC uses the standard Structured Query Language (SQL) for all database interaction.
- Does not abstract the database, does not hide the database structure
- Almost all JDBC functionality is accomplished through sending SQL statements to the database server

JDBC vs ADO.NET

- ADO.NET is the .NET counterpart to JDBC, providing a similar API for database access
- ADO.NET uses **data providers** (similar idea with the JDBC drivers, different terminology)
- Both JDBC and ADO.NET use a model where the application sends SQL as a string to the database, and the database executes it:

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
```

```
SqlCommand cmd = new SqlCommand("SELECT * FROM users", conn);
SqlDataReader reader = cmd.ExecuteReader();
```

- ADO.NET has more features: supports both connected (DataReader) and disconnected (DataSet/DataTable) models, allowing data to be processed even after the database connection is closed

Problem: Boilerplate Code

- Working with JDBC or ADO.NET = a lot of boilerplate code:
  - Obtaining connection
  - SQLException handling
  - ***creating objects out of the query results***:

```
ResultSet rs = ...
while(rs.next()) {
    Person p = new Person();
    p.setId(rs.getLong("ID"));
    p.setName(rs.getString("NAME"));
}
```

- Solution: Object Relational Mapping (ORM)
- ORM basic idea: map whole classes directly to database records
- JDBC -> JPA, Hibernate
- ADO.NET -> Entity Framework Core, Dapper

Problem: Abstraction, Separation of Concerns

- Using directly JDBC / ADO.NET mixes Business logic and Database access logic

- Solution: different pattern for decoupling
- DAO (Data Access Object pattern) - encapsulates all database access in a separate layer
- Java:
  - Manual implementation of DAO pattern for data access
  - Spring Data JPA - Automatic generated DAO (Auto=generated Repository in Spring Data)
- .NET:
  - Repository pattern