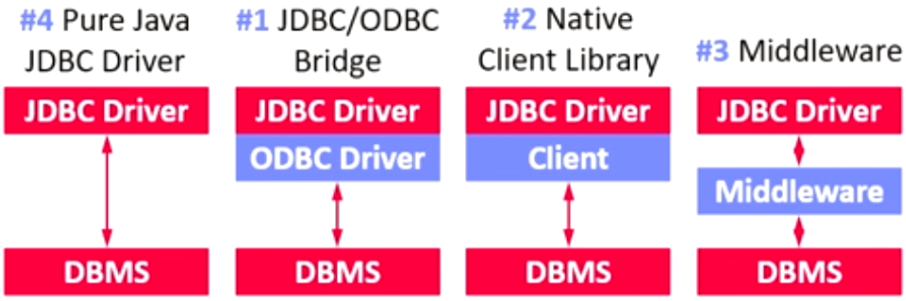
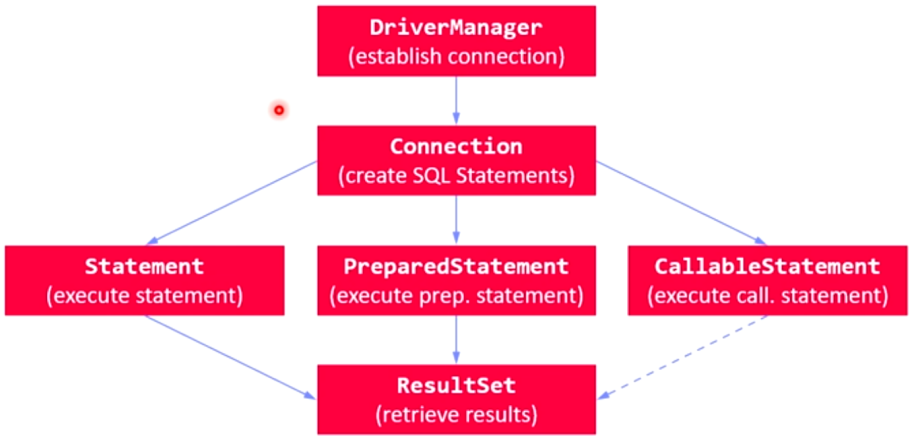
Overview

- Java Database Connectivity
- [[Call Level Interfaces]] for accessing databases independent of DBMS from Java
- most relational DBMS have JDBC implementations
- driver types
  - 

JDBC Components and Flow

- driver manager establishes connection to execute (prepared/callable) statements which may return results
  - 

- prepared statements
  - avoid [[SQL Injection]] because inserted data need specific datatypes
    - regular statements just execute query strings
  - reusable ==> better performance
- callable statements
  - prepared statement which calls a stored

Example

- **Establishing a Connection**
  - DBMS-specific URL strings including host, port, and database name
  - Stateful handles representing user-specific DB sessions
  - JDBC driver is usually a jar on the class path
  - Connection and statement pooling for performance

```
Connection conn = DriverManager
   .getConnection("jdbc:postgresql:"+
   "//localhost:5432/db1234567",
   username, password);
```

```
META-INF/services/
java.sql.Driver
```

- **Execute `Statement`**
  - Use for simple SQL statements w/o parameters
  - Beware of SQL injection
  - API allows fine-grained control over fetch size, fetch direction, batching, and multiple result sets

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql1);
...
int rows = stmt.executeUpdate(sql2);
stmt.close();
```

Note: PostgreSQL does not support fetch size but sends entire result

- **Process ResultSet**
  - Iterator-like cursor (app-level) w/ on-demand fetching
  - Scrollable / updatable result sets possible
  - Attribute access via column names or positions

```
ResultSet rs = stmt.executeQuery(
   "SELECT SID, LName FROM Students");

List<Student> ret = new ArrayList<>();
while( rs.next() ) {
   int id = rs.getInt("SID");
   String name = rs.getString("LName");
   ret.add(new Student(id, name));
}
```

- **Execute `PreparedStatement`**
  - Use for precompiling SQL statements w/ input params
  - Inherited from Statement
  - Precompile SQL once, and execute many times

  → Performance

  → No danger of SQL injection

```
PreparedStatement pstmt =
   conn.prepareStatement(
     "INSERT INTO Students VALUES(?,?)");

for( Student s : students ) {
   pstmt.setInt(1, s.getID());
   pstmt.setString(2, s.getName());
   pstmt.executeUpdate();
}

pstmt.close();
```

- **Null Handling**
  - Pass null object (explicitly for primitive types)

```
pstmt.setString(2, p[1]);
pstmt.setObject(3, p[2].isEmpty() ?
    null : Integer.valueOf(p[2]),
    Types.INTEGER);
```

- **Queries and Updates**
  - Queries → `executeQuery()`
  - Insert, delete, update → `executeUpdate()`

- **Recap: (Stored Procedures, see 05 Query Languages (SQL))**
  - Can be **called standalone via CALL** <proc_name>(<args>);
  - Procedures return no outputs, but might have **output parameters**

- **Execute `CallableStatement`**
  - Create prepared statement for call of a procedure
  - Explicit registration of output parameters
  - Example

```
CallableStatement cstmt = conn.prepareCall(
   "{CALL prepStudents(?, ?)}");

cstmt.setInt(1, 2019);
cstmt.registerOutParameter(2, Types.INTEGER);
cstmt.executeQuery();

int rows = cstmt.getInt(2);
```

Transaction Handling

- [[Transaction]] disabled by default
- can be enabled
- transactions can be fully committed or rolled back in case an error occurs

- **JDBC Transaction Handling**
  - Isolation levels (incl NONE) and (auto) commit option
  - Savepoint and rollback (undo till begin or savepoint)
  - **Note:** TX handling on connection not statements

- **Beware of Defaults**
  - DBMS-specific default isolation levels

(SQL Standard: **SERIALIZABLE**, PostgreSQL: **READ COMMITTED**)

```
conn.setTransactionIsolation(
   TRANSACTION_SERIALIZABLE);
conn.setAutoCommit(false);

PreparedStatement pstmt = conn
   .prepareStatement("UPDATE Account
   SET Balance=Balance+? WHERE AID = ?");

Savepoint save1 = conn.setSavepoint();

pstmt.setInt(1,-100); pstmt.setInt(107);
pstmt.executeUpdate();

if( rand()<0.1 )
   conn.rollback(save1);

pstmt.setInt(1,100); pstmt.setInt(999);
pstmt.executeUpdate();

conn.commit();
```

- batch inserts/fewer commits can increase performance

```
conn.setAutoCommit(false);
PreparedStatement pstmt = conn.prepareStatement(
  "INSERT INTO Persons(AKey, Name, Website, IKey) VALUES(?,?,?,?)");
for( String[] p : tmp ) {
  pstmt.setInt(1, Integer.valueOf(p[0].substring(1)));
  pstmt.setString(2, p[1]);
  pstmt.setString(3, p[5].isEmpty() ? null : p[5]);
  pstmt.setObject(4,
    orgs.get(p[3]+"_"+p[4]),
    Types.INTEGER);
  pstmt.executeUpdate();
}
conn.commit();
```

Performance Ref Implementation SS2020:
(36K authors, 28K papers, 101K author-papers)

```
* Auto Commit:        23.7s
* Batched Commits:    12.5s
```

Performance Ref Implementation SS2021:
(116K athletes, 158K team-athletes, 219K results)

```
* Auto Commit:        68.7s
* Batched Commits:    36.3s
```