Java Database Connectivity (JDBC)

History - JDBC 1

- Java API clients for the implementation of database access
- 1997: JDK 1.1 – JDBC 1.2
  - classes111.jar, classes111_g.jar, nls_charset11.jar
  - Core API: `java.sql`
  - Main interfaces and classes: Driver, DriverManager, Connection, Statement, PreparedStatement, CallableStatement, ResultSet, DatabaseMetaData, ResultSetMetaData, Types

History - JDBC 2

- 1999: JDK 1.2 – JDBC 2.1
  - classes12.jar, classes12_g.jar, ocrs12.jar, nls_charset12.jar
  - Extension of Core API
    - Extending existing interfaces with new features, better performance, and SQL3 data types
      - Scrollable result sets, programming support for insertion, deletion and updating, etc.
  - Optional Package API: `javax.sql`
    - DataSource: cooperation with JNDI
    - Connection pooling: reuse of connections
    - Distributed transactions: usage of multiple databases in a transaction
    - Rowset: JavaBean component representing a relation

History - JDBC 3

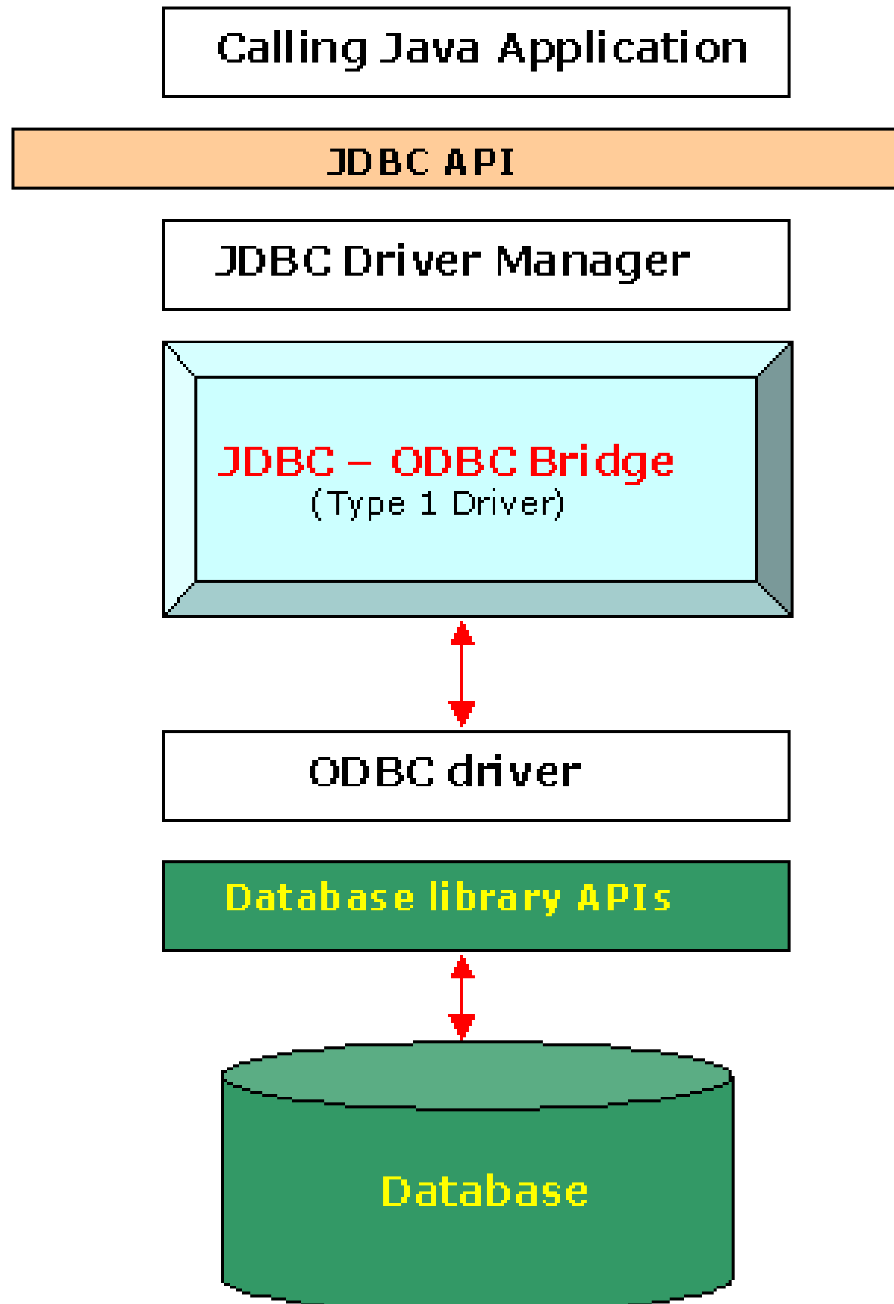
- 2001: JDK 1.4 – JDBC 3.0
  - ojdbc14.jar, ojdbc14_g.jar, ocrs12.jar
  - Savepoints
  - Connection pooling fixes (refinement)
  - Statement pooling
  - Query of automatically generated keys
  - BLOB/CLOB modification
  - Mutliple open result sets
  - Other, e.g. query of parameter metadata, binding of parameters of stored subprograms by name, resultset holdability

History - JDBC 4

- 2006: Java SE 6 – JDBC 4.0
  - ojdbc6.jar, ojdbc6_g.jar, ojdbc6dms.jar, and ojdbc6dms_g.jar
  - Does not support the oracle.jdbc.driver package
    - Instead: oracle.jdbc
  - Does not support Java versions before J2SE 5.0
  - Other safety properties: AES, SHA1, Radius, Kerberos, SSL
  - SQL ANYDATA and ANYTYPE support
  - Database initialization and shutdown support
  - Better memory management and performance

Driver

- The software component that allows the Java application to establish contact to a database
- It has 4 types according to the JDBC specification


Calling Java Application
JDBC API
JDBC Driver Manager
JDBC – ODBC Bridge
(Type 1 Driver)
ODBC driver
Database library APIs
Database

Type 1
JDBC-ODBC bridge

- Properties
  - JDBC requests turned into ODBC requests, which will be handled by the OBDC-driver
  - Client → JDBC driver → ODBC driver → database
- Advantages
  - Almost every database with an ODBC-driver can be reached
  - Ease of installation
- Disadvantages
  - Performance loss
  - The ODBC-driver has to be installed on the client
  - Inappropriate for applets/internet applications

Calling Java Application

JDBC API

JDBC Driver Manager

Native-API driver

(Type 2 Driver)

Database library APIs

Database

Type 2
Native API driver

- Properties
  - Client calls the API of the database
  - Client → JDBC driver → client side library of vendor → database
- Advantages
  - Better performance (no JDBC-ODBC translation)
- Disadvantages
  - Vendor's library has to be installed on client
  - Thus inappropriate for web apps
  - Not all databases have client side libraries

Calling Java Application

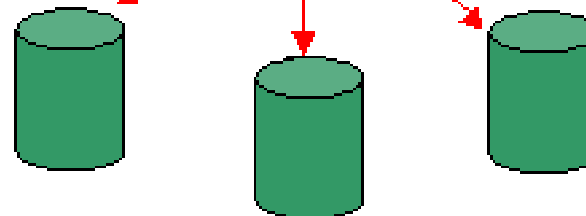
JDBC API

JDBC Driver Manager

Network-Protocol driver
(Type 3 Driver)

Middleware

(Application server)



Different database vendors

Type 3
Network protocol driver

- Properties
  - Written only in Java
  - Three layer communication
  - Can be used for multiple databases (vendor independent)
  - Communication between the client and the middle layer is database independent
  - The middle layer translates to the language of the database
  - Client → JDBC driver → intermediate layer server → database
- Advantages
  - No need for library from vendor on client side
  - Changing between databases does not affect the connection between the client and the middle layer
  - The middle layer can help work with typical services such as caching, load balancing, logging, auditing, etc.
- Disadvantages
  - Database specific coding required on the middle layer
  - The new layer can prove to be a bottleneck regarding time

Calling Java Application

JDBC API

JDBC Driver Manager

Native-Protocol driver
(Type 4 Driver)

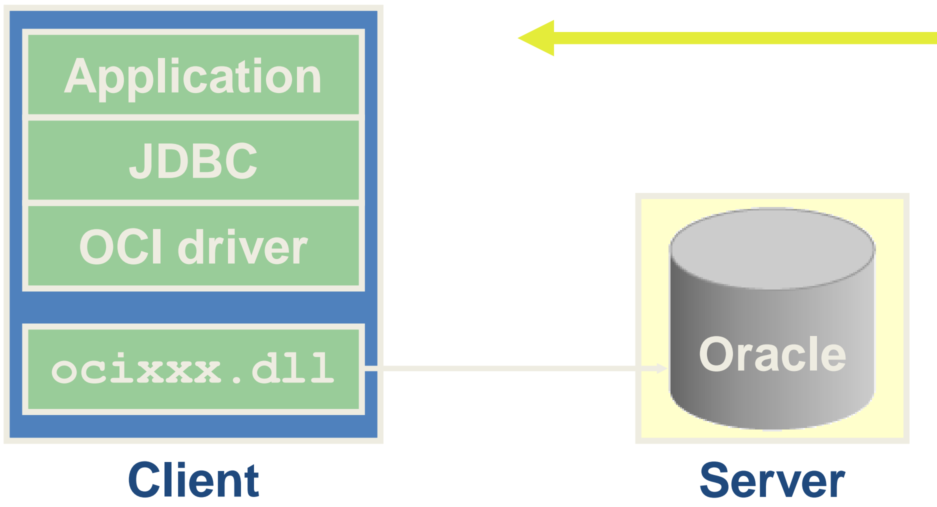
direct calls using
specific database protocol

Type 4
Native-protocol driver

- Properties
  - Written clearly in Java, direct communication with the databsae (socket)
  - The driver forms the JDBC calls to be appropriate for the vendor specific database protocol
  - Client → native protocol JDBC driver → database
- Advantages
  - No intermediate format, no intermediate layer – better performance
  - All aspects of the connection between the application and the database are handled in the JVM – easier debugging
- Disadvantages
  - All databases need different drivers on the client side

Oracle and JDBC

- Oracle provides 2 client side JDBC-drivers
- Oracle provides 2 server side JDBC-drivers

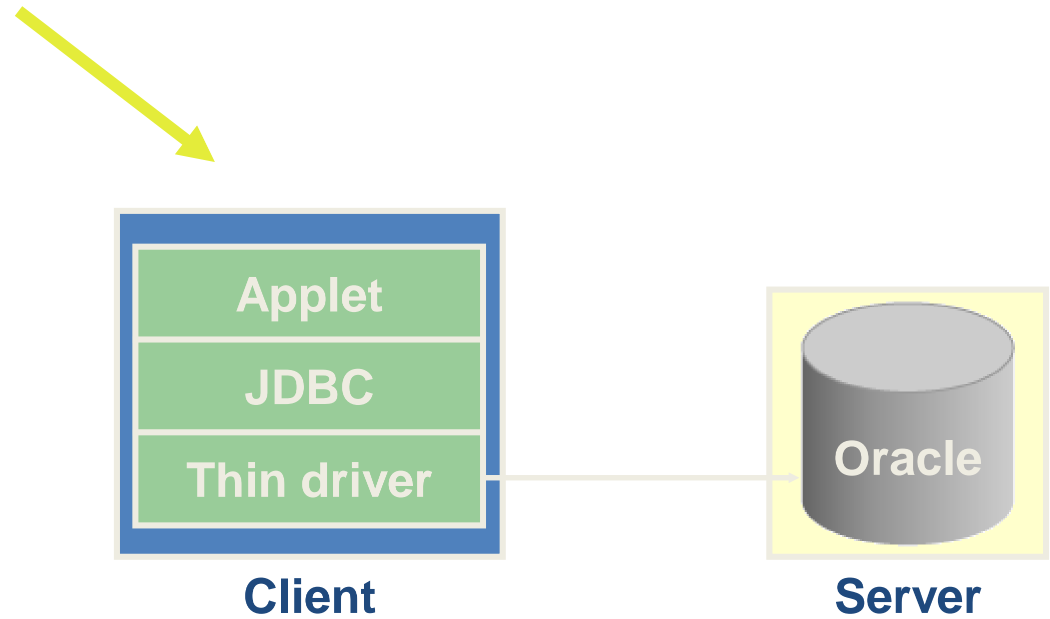


JDBC OCI client side driver (thick driver)

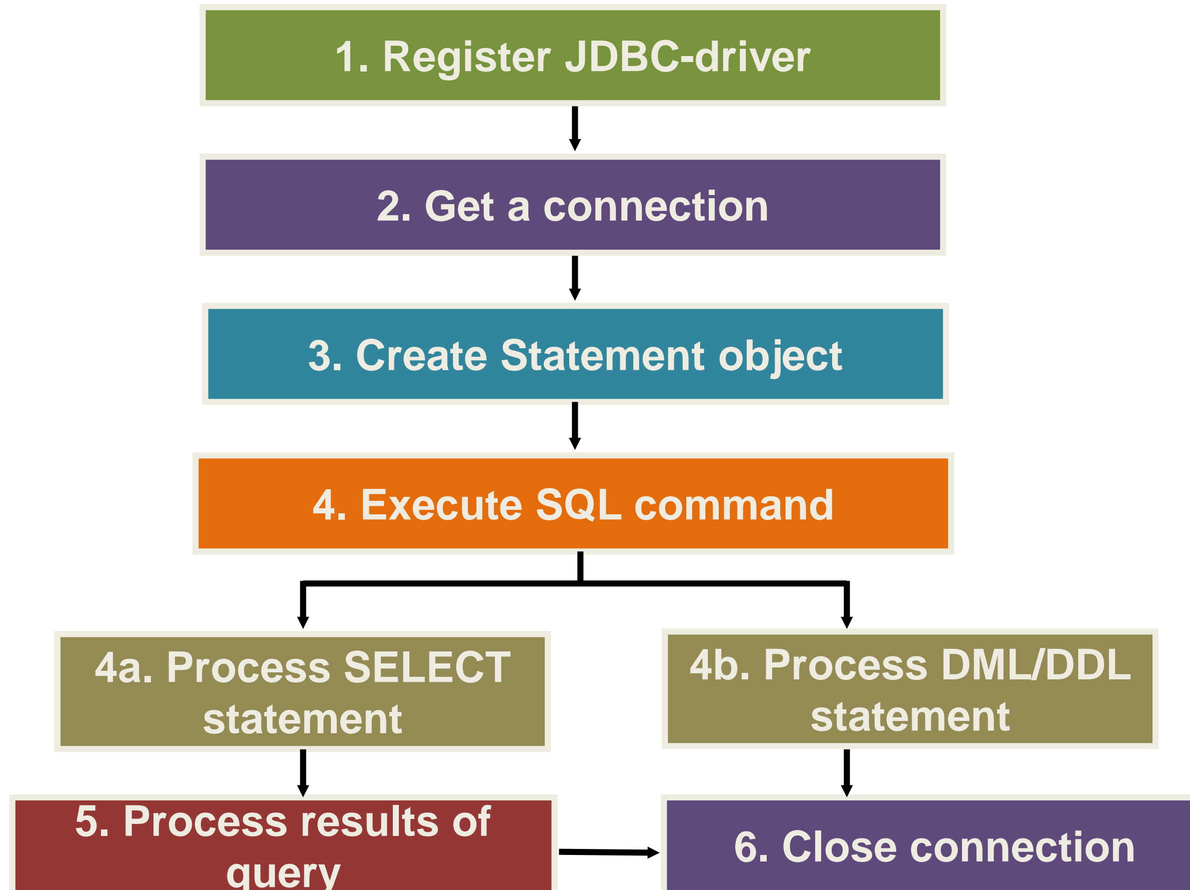
Type 2, through Oracle Call Interface
Oracle Client has to be installed on client, only used by applications (not applets)

JDBC thin client side driver (thin driver)

Type 4, through TCP/IP
Client only needs the driver, can be used from applets and Applications as well



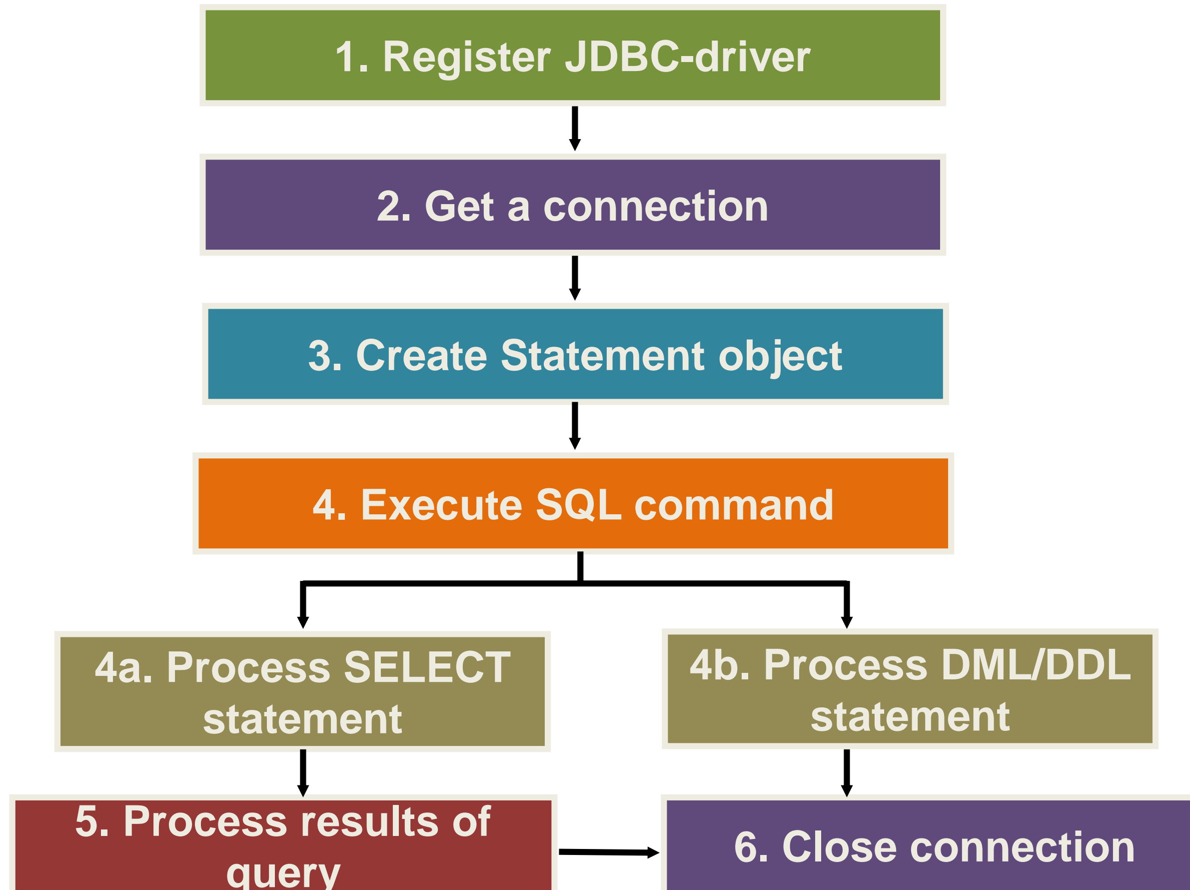
Steps of executing SQL commands



Step 1: registering the driver

- In code:
  - ```
    DriverManager.registerDriver (new oracle.jdbc.OracleDriver());
    ```
  - ```
    Class.forName ("oracle.jdbc.OracleDriver");
    ```
- When loading the class:
  - ```
    java -D jdbc.drivers = oracle.jdbc.OracleDriver <ClassName>;
    ```
    ```

Steps of executing SQL commands

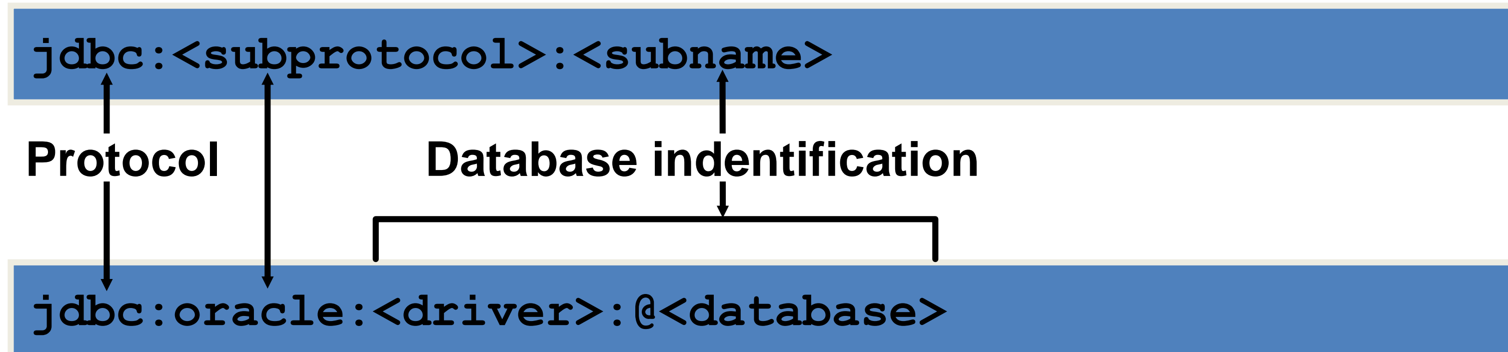


Step 2: getting a database connection

- JDBC 1.0: DriverManager.getConnection()

```
Connection conn = DriverManager.getConnection(
"jdbc:oracle:thin:@db.inf.unideb.hu:1521:ora11g",
      "user","passwd");
```

- JDBC URL structure



JDBC URLs in Oracle

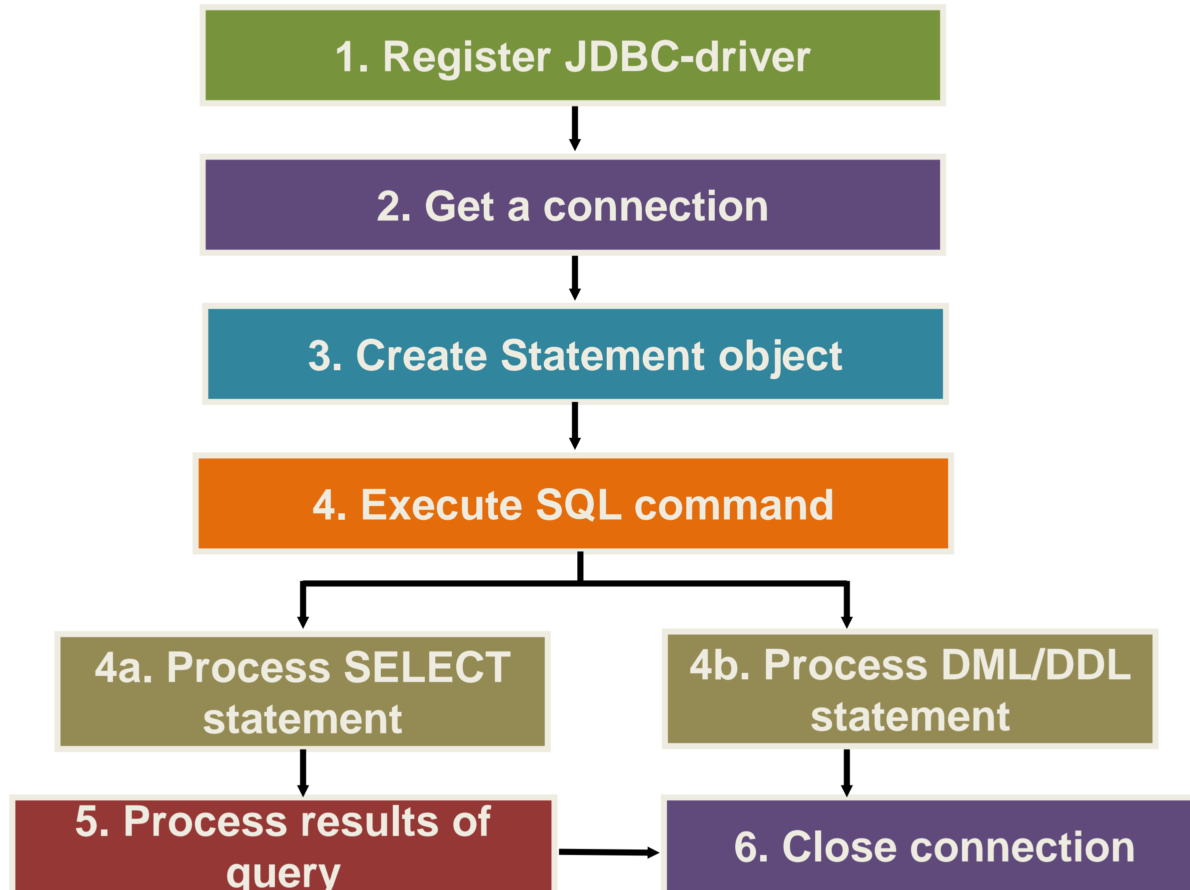
– Oracle Thin driver

```
Syntax:   jdbc:oracle:thin:@<host>:<port>:<SID>
Example: "jdbc:oracle:thin:@myhost:1521:orcl"
```

– Oracle OCI driver

```
Syntax:   jdbc:oracle:oci:@<tnsname entry>
Example: "jdbc:oracle:oci:@orcl"
```

Steps of executing SQL commands



Step 3: Creating a Statement object

- Based on the Connection instance

```
Statement stmt = conn.createStatement();
```

- Methods of the Statement interface:
  - For `SELECT` execution:

    `ResultSet executeQuery(String sql);`

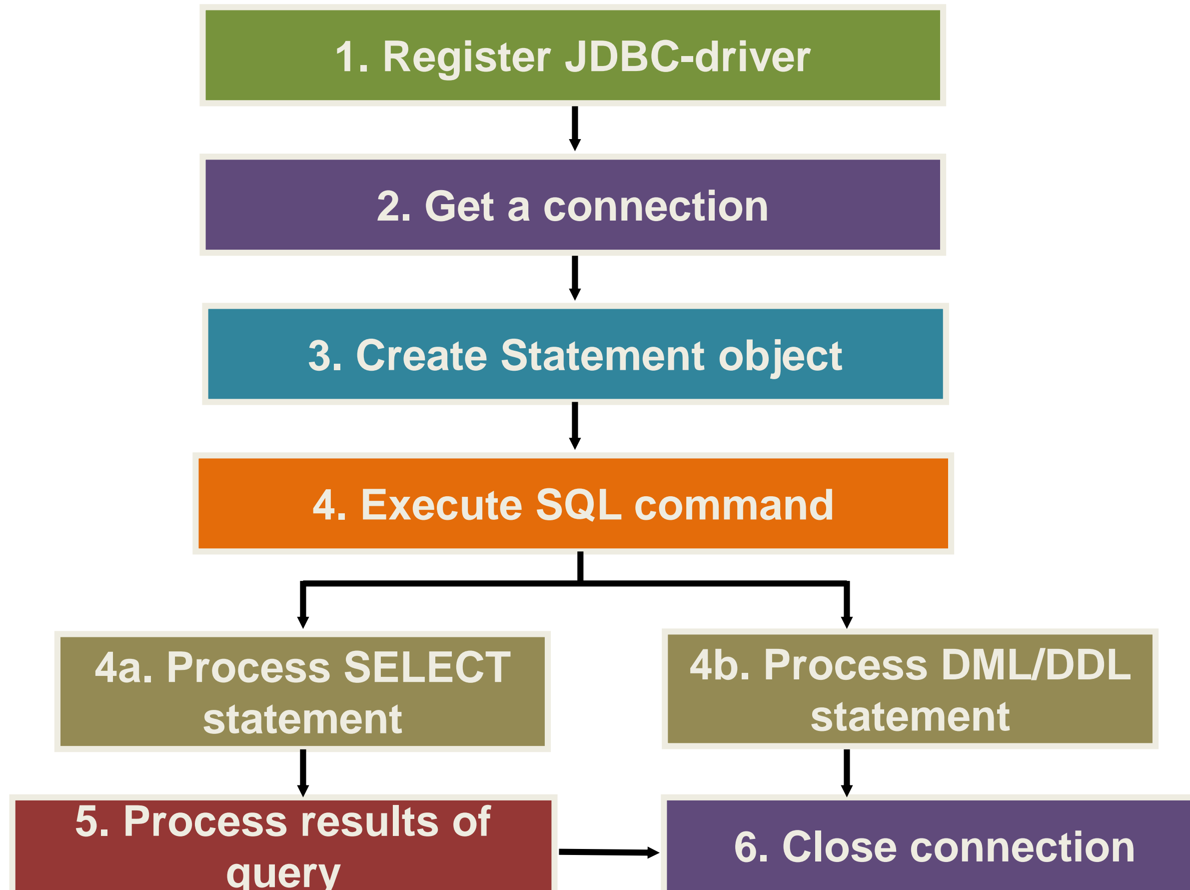
  - For execution of other DML or DDL commands:

    `int executeUpdate(String sql);`

  - For executing arbitrary SQL commands:

    `boolean execute(String sql);`

Steps of executing SQL commands



Step 4a: Executing a query

- Define the text of the query (without semicolon!)

```
ResultSet rset = stmt.executeQuery
       ("SELECT ename FROM emp");
```

- The ResultSet interface
  – Represents the resultset of a query (the resulting relation)
  – Shows the actual row with a cursor (its initial position is before the first row)
  – Cannot be modified by default, and can only be navigated forward, can only be went through once
    - Rewritable since JDBC 2

Step 5: Going through a resultset

- With the next() and getXXX() methods of ResultSet

```
while (rset.next())
   System.out.println (rset.getString(1));
```

  - Potencial problem: NULL and getInt()
    - solution: use getBigDecimal()!
- L

```
rset.close();
stmt.close();
```

Step 4b: executing a DML command

```
int rowcount = stmt.executeUpdate
   ("DELETE FROM order_items
       WHERE order_id = 2354");
```

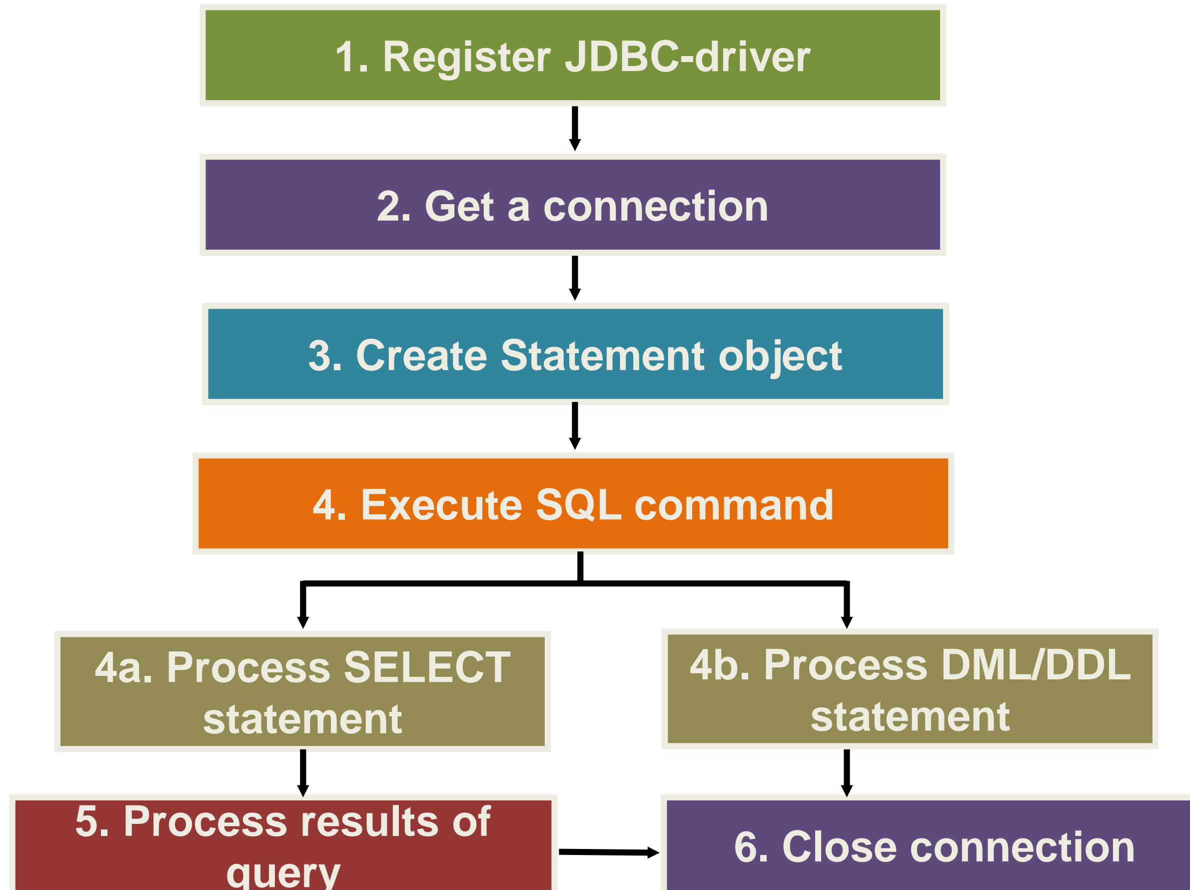
```
int rowcount = stmt.executeUpdate (
      "INSERT INTO pictures (id) " +
            "SELECT region_id FROM regions");
```

```
int rowcount = stmt.executeUpdate
   ("UPDATE employees SET salary = 1.1 * salary
WHERE dept_id = 20");
```

Step 4b: executing a DDL command

```
int rowcount = stmt.executeUpdate
  ("CREATE TABLE temp (col1 NUMBER(5,2),
      col2 VARCHAR2(30))");
```

Steps of executing SQL commands

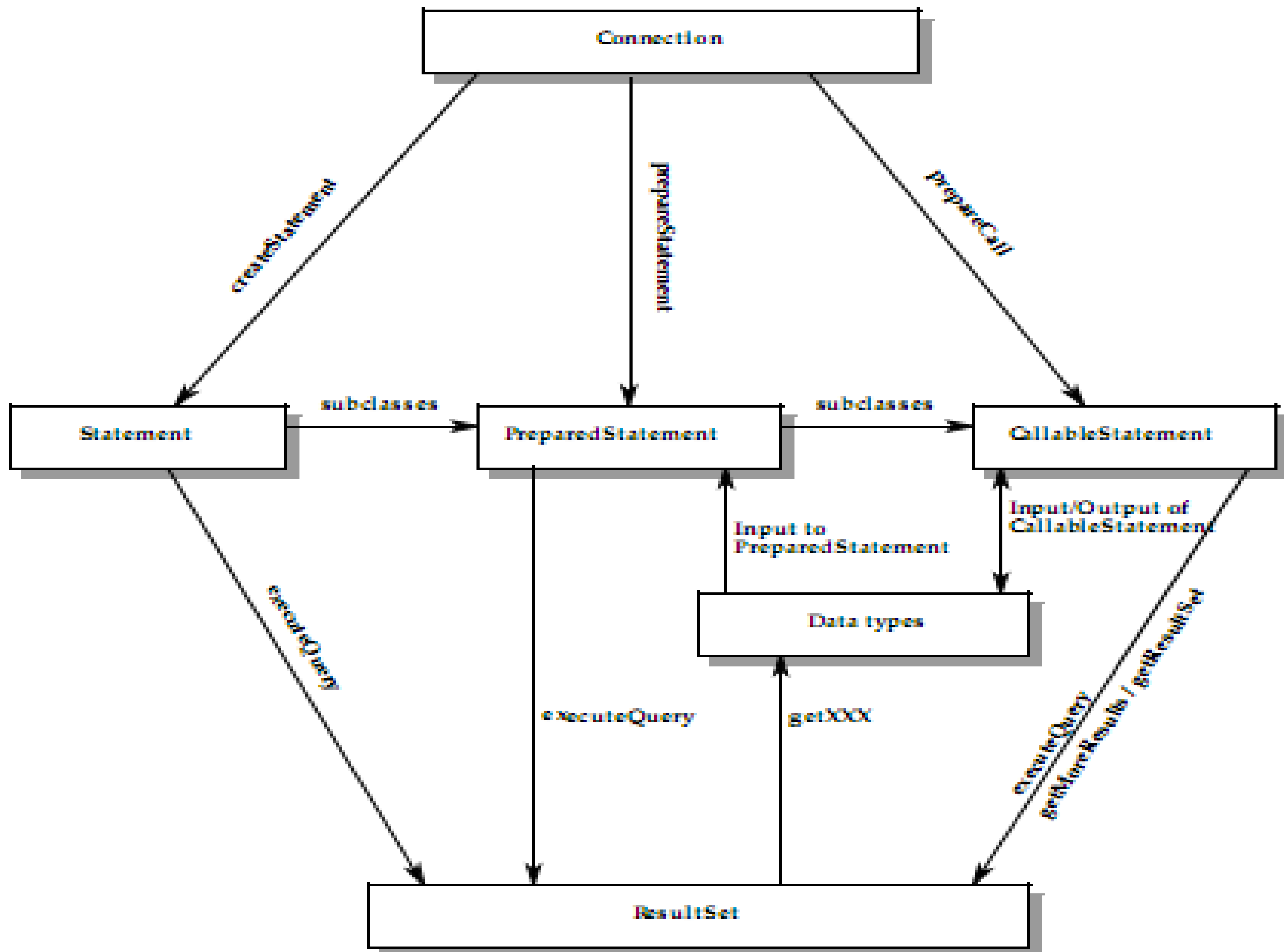


Step 6: closing the connection

```
Connection conn = ...;
Statement stmt = ...;
ResultSet rset = stmt.executeQuery(
          "SELECT ename FROM emp");
...
// clean up
rset.close();
stmt.close();
conn.close();
...
```

Handling unknown commands

```
boolean isQuery = stmt.execute(SQLstatement);
if (isQuery) { // query
  ResultSet r = stmt.getResultSet(); ...
}
else { // other DML or DDL command
  int count = stmt.getUpdateCount(); ...
}
```



Connection
createStatement
prepareStatement
prepareCall
Statement
subclasses
PreparedStatement
subclasses
CallableStatement
Input to
PreparedStatement
Input/Output of
CallableStatement
Data types
executeQuery
executeQuery
getXXX
executeQuery
getMoreResults / getResultSet
ResultSet

PreparedStatement

- Derived from Statement; for storing precompiled SQL commands
- Used when a command is to be executed multiple times
- Parametrizable: actual parameters have to be defined when executed

PreparedStatement

After registering the driver and establishing the connection:

```
PreparedStatement pstmt =
  conn.prepareStatement
("UPDATE emp SET ename = ? WHERE empno = ?");
```

```
PreparedStatement pstmt =
  conn.prepareStatement
("SELECT ename FROM emp WHERE empno = ?");
```

```
pstmt.setXXX(index, value);
```

```
pstmt.executeQuery();
pstmt.executeUpdate();
```

PreparedStatement - example

After registering the driver and establishing the connection:

```
int empNo = 3521;
PreparedStatement pstmt = conn.prepareStatement(
   "UPDATE emp SET ename = ? WHERE empno = ? ");
pstmt.setString(1, "DURAND");
pstmt.setInt(2, empNo);
pstmt.executeUpdate();
```

- Setting a NULL value:

```
pstmt.setNull(1, java.sql.Types.VARCHAR);
```

ResultSetMetaData

```
PreparedStatement pstmt =
  conn.prepareStatement(
                    "SELECT * FROM CATALOG");
ResultSetMetaData rsmd = pstmt.getMetaData();
int colCount = rsmd.getColumnCount();
int colType;
String colLabel;
for (int i = 1; i <= colCount; i++) {
  colType = rsmd.getColumnType(i);
  colLabel = rsmd.getColumnLabel(i);
  ...
}
```

Exception handling

```
 try  {
      rset = stmt.executeQuery("SELECT empno, name
FROM emp");
}
        catch (java.sql.SQLException e)
        { ... /* handle SQL errors */ }
...
        finally {  // clean up
        try {
          if (rset != null) rset.close();
        } catch (Exception e)
         { ... /* handle closing errors */ }
...
```

Transaction handling

Autocommit mode is enabled by default

This can be disabled with a `conn.setAutoCommit(false)` call

In the latter case

- `conn.commit()`: commits
- `conn.rollback()`: rolls back

The closing of the connection means commiting