

Database Programming Approaches

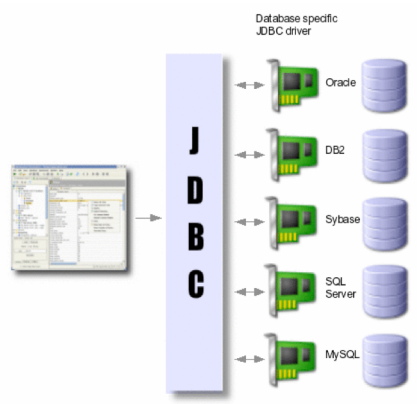
- ❑ Embedded commands:
 - Database commands are **embedded** in a general-purpose programming language
- ❑ Library of database functions:
 - Available to the host language for database calls; known as an **API** (Application Program Interface)
 - e.g., *JDBC, ODBC, PHP*
- ❑ A brand new, full-fledged language
 - e.g., Oracle PL/SQL
 - **Procedural Language extensions to SQL**

JDBC: An example of SQL API

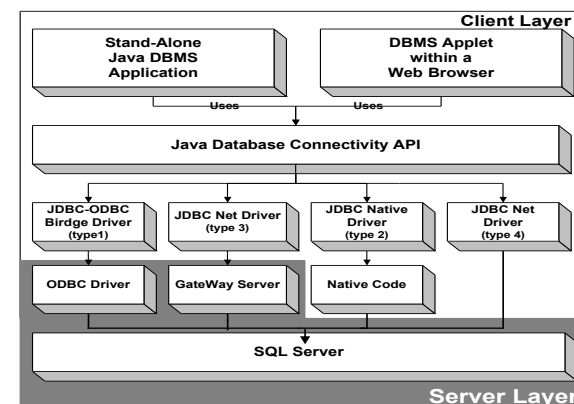
- ❑ JDBC resembles dynamic SQL, in which SQL statements are passed in the form of strings
- ❑ JDBC supports its own dialect of SQL
- ❑ An application program (Java applet) executes an SQL statement by submitting it to the JDBC driver manager
- ❑ Any database using can be accessed as long as an appropriate DBMS-specific driver exists, is loaded, and is registered with the driver manager:

```
import java.sql.*;
Class.forName("jdbc.driver_name");
```

JDBC Configuration



JDBC Drivers



JDBC Drivers

- ❑ Type 1: JDBC-ODBC bridge.
This driver translates JDBC calls into ODBC calls.
- ❑ Type 2: JDBC-Gateway.
This pure Java driver connects to a database middleware server that in turn interconnects multiple databases and performs any necessary translations.
- ❑ Type 3:
Java JDBC Native Code. This partial Java driver converts JDBC calls into client API for the DBMS.
- ❑ Type 4:
Pure Java JDBC. This driver connects directly to the DBMS.

Accessing a Database

- ❑ Open Connection:
`Connection dbcon;`
`dbcon=`
`DriverManager.getConnection(<"URL">,<"userId">,<"pwd">);`
- ❑ E.g.,

```
import java.sql.*;
public class JavaDemo {
    private Connection dbcon; private String username = "PittID", password = "PSNum";
    public JavaDemo() {
        DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
        String url = "jdbc:oracle:thin:@class3.cs.pitt.edu:1521:dbclass";
        dbcon = DriverManager.getConnection(url, username, password);
        ...
    }
}
```
- ❑ Close connection: `dbcon.close();`

Executing an SQL Statement

- ❑ Statement class: Execute SQL statements without parameters
 - Create statement object
`Statement st;`
`st = dbcon.createStatement();`
 - Directly execute: Select, Update, Insert, Delete, DDL
`st.executeQuery(<"sql-query">);`
`st.executeUpdate(<"sql-modification">);`
- ❑ Example of an SQL modification
`int numberrows = st.executeUpdate`
`("INSERT INTO STUDENT VALUES (123, 'J.J. Kay', 'CS')");`

Querying a database & Cursors

```
String fname = readString("Enter First Name: ");
String query1 = "SELECT SID, Name, Major FROM STUDENT
                WHERE Name LIKE '" + fname + "'";
ResultSet res1 = st.executeQuery(query1);

int rsid; String mname, rmajor;
while (res1.next()) {
    rsid = res1.getInt("SID");
    mname = res1.getString("Name");
    rmajor = res1.getString(3);

    System.out.print(rsid+" "+mname+" "+rmajor);
};
```

`getXXX(param)`
XXX: valid SQL Type
param: name or index

`wasNULL()` returns True
if the last `getXXX()` value
should be read as NULL

Cursor Positioning

```
Statement stC = dbcon.createStatement
    (ResultSet.TYPE_SCROLL_INSENSITIVE,
     ResultSet.CONCUR_READ_ONLY);
ResultSet resultSet = stC.executeQuery("SELECT * FROM STUDENT");
int pos = resultSet.getRow();           // Get cursor position, pos = 0
boolean b = resultSet.isBeforeFirst(); // true
resultSet.next();                       // Move cursor to the first row
pos = resultSet.getRow();               // Get cursor position, pos = 1
b = resultSet.isFirst();               // true
resultSet.last();                      // Move cursor to the last row
pos = resultSet.getRow();               // If table has 10 rows, pos = 10
b = resultSet.isLast();                // true
resultSet.afterLast();                 // Move cursor past last row
pos = resultSet.getRow();               // If table has 10 rows, value would be 11
b = resultSet.isAfterLast();           // true
```

Cursor Navigation Types

- ❑ Statement stC = dbcon.createStatement
({ResultSet.TYPE_XXXX});
- ❑ TYPE_XXXX
 - TYPE_FORWARD_ONLY: ResultSet can only be navigated forward.
 - SCROLL_INSENSITIVE: ResultSet can be navigated forward, backwards and jump. Concurrent db changes are not visible.
 - SCROLL_SENSITIVE: ResultSet can be navigated forward, backwards and jump. Concurrent db changes are visible.

Cursor Concurrency Types

- ❑ Statement stC = dbcon.createStatement
(ResultSet.TYPE_XXXX);
- ❑ TYPE_XXXX
 - CONCUR_READ_ONLY: ResultSet can only be read
 - CONCUR_UPDATABLE: ResultSet can be updated

The PreparedStatement Class

- ❑ Create and pre-compile parameterized queries using parameters markers, indicated by question marks (?)

```
PreparedStatement st2 = dbcon.prepareStatement
    ("SELECT * FROM STUDENT WHERE Name LIKE ?");
```
- ❑ Specify the values of parameters using setXXX(i,v)
where XXX: SQL type in including NULL,
i: argument-index,
v: value

```
String fname = readString("Enter First Name: ");
st2.setString(1, fname);
ResultSet res2 = st2.executeQuery();
```

Querying the Catalog & Native SQL

Metadata about results

```
ResultSet res3 = st.executeQuery("SELECT * FROM STUDENT");
ResultSetMetaData resmetadata = res3.getMetaData();
int num_columns = resmetadata.getColumnCount();
string column_name = resmetadata.getColumnName(3);
```

Metadata about database

```
DatabaseMetaData dbmd = dbcon.getMetaData();
```

Native SQL

```
nativeSQL(String sql);
    ■ Converts SQL stmt into the system's native SQL grammar
```

Error Handling

JDBC provides the SQLException class to deal with errors

```
try { ResultSet res3 =
    st.executeQuery("SELECT * FROM STUDENT"); }
catch (SQLException e1) {
    System.out.println("SQL Error");
    while (e1 != null) {
        System.out.println("Message = "+ e1.toString());
        System.out.println("SQLState = "+ e1.getSQLstate());
        System.out.println("SQLState = "+ e1.getErrorCode());
        e1 = e1.getNextException();
    };
};
```

Executing Transactions

Each JDBC statement is treated as a separate transaction that is autocommitted by default

```
dbcon.setAutoCommit(false);
```

A new transaction automatically is set after either

```
dbcon.commit; or dbcon.rollback;
```

Set Constraint Mode

```
ResultSet res1 = st.executeQuery("SET CONSTRAINTS ALL DEFERRED");
```

Five transaction isolation levels (to be discussed later)

```
■ setTransactionIsolation(int level);
```

No global transactions, transactions across many db

```
■ No atomicity or "all or nothing property"
```

Not Deferred Constraints

Transaction atomicity is enforced in a flexible way by the developer (with the support of the DBMS), e.g.:

```
try {
    dbcon.setAutoCommit(false);
    st.executeUpdate("insert into student values (23, 'John', 'CS')");
    st.executeUpdate("insert into Dept values (15, 'Joanne', 'CoE')");
    dbcon.commit();
}
catch (SQLException e1) {
    try {
        dbcon.rollback();
    }
    catch (SQLException e2) { System.out.println(e2.toString()); }
}
```

SQL injection vulnerabilities

- ❑ Allow an attacker to inject (or execute) SQL commands within an application
- ❑ Typical example:

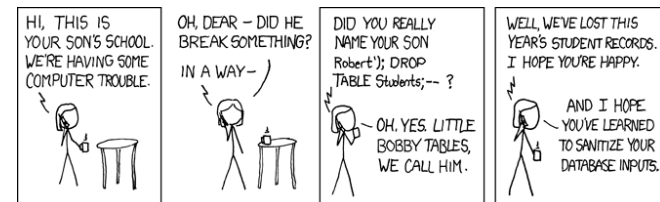
```
Connection dbcon = db.getConnection( );
String sql = "SELECT * FROM user WHERE
             username= " + username + " and password=" + password +
             """;
Statement stmt = dbcon.createStatement();
int rs = stmt.executeQuery(sql);
if (rs.next()) { loggedIn = true; out.println("Successfully logged in"); }
else { out.println("Username and/or password not recognized"); }
```

What is the problem?

- ❑ Accepting user input without performing adequate input validation or escaping meta-characters
- ❑ Example inputs:
admin' (for username)
'1' OR '1'='1. (for password)
- ❑ Result:
SELECT * FROM user
WHERE username='admin' and password='1' OR '1'='1';
- ❑ Effect: ?

What is the problem?

- ❑ String sql = "SELECT * FROM user WHERE
username= " + username + " and password=" + password + """;
- ❑ Example inputs:
panos'
password = '3113'; DELETE FROM user WHERE '1'
- ❑ Result:
SELECT * FROM user
WHERE username='panos' and password='3113';
DELETE FROM user WHERE '1';
- ❑ Effect: ?



Avoiding SQL Injection

- ❑ In the same way attackers can inject other SQL commands
 - extract, update or delete data within the database
- ❑ Solution: Good programming practice; use `prepareStatement()`
 - All queries should be parameterized
 - All dynamic data should be explicitly bound to parameterized queries
 - String concatenation should never be used to create dynamic SQL (in general)

- ❑ Example:

```
PreparedStatement st3 = dbcon.prepareStatement  
("SELECT * FROM user WHERE username= ? AND password = ?");
```