Cursor Navigation Types

- Statement stC = dbcon.createStatement ([{ResultSet.NAV_TYPE}], [ResultSet.CONCUR_TYPE]);
- NAV_TYPE
  - 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.NAV_TYPE}], [ResultSet.CONCUR_TYPE]);
- CONC_TYPE
  - CONCUR_READ_ONLY: ResultSet can only be read
  - CONCUR_UPDATABLE: ResultSet can be updated

Moving Cursors

```
Statement stC = dbcon.createStatemen
                    (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
```

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();
```

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.getMessage());
        System.out.println("SQLState = "+ e1.getSQLstate());
        System.out.println("SQLState = "+ e1.getErrorCode());
        e1 = e1.getNextException();
     }; };
```

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()); }
   }
```

JDBC: Stored Functions

- SQL:
```
SELECT upper('database'); -- Invocation in DataGrip
```

- **"select"** for functions only (default)
- **"callIfNoReturn"** for procedures and functions
- **"call"** for procedures only

- JAVA:
```
props.setProperty("escapeSyntaxCallMode", "callIfNoReturn");
Connection conn = DriverManager.getConnection(url, props);

CallableStatement upperFunc =
                 conn.prepareCall("{? = call upper( ? ) }");
upperFunc.registerOutParameter(1, Types.VARCHAR);
upperFunc.setString(2, "database");
upperFunc.execute();

String upperCased = upperFunc.getString(1);
upperFunc.close();

System.out.println(upperCased);
```

JDBC: Stored Functions

- SQL:
```
CREATE OR REPLACE FUNCTION fx()
RETURNS SETOF student AS $$
BEGIN
  RETURN QUERY SELECT * FROM student;
END
$$ LANGUAGE plpgsql;

SELECT * FROM fx(); -- Invocation in DataGrip
```

- JAVA:
```
ResultSet res1 = st.executeQuery("SELECT * FROM fx()");
int rid;
String rname, rmajor;
while (res1.next()) {
 rid = res1.getInt("SID");
 rname = res1.getString("QPA");
 rmajor = res1.getString(3);
 System.out.println(rid + " " + rname + " " + rmajor);
}
```

JDBC: Stored Procedures

- SQL:
```
CREATE OR REPLACE PROCEDURE
change_major_proc(varchar(5),varchar(5))
     LANGUAGE plpgsql AS $$
BEGIN
   UPDATE STUDENT SET MAJOR=$2 WHERE Major=$1;
END$$;

begin; -- Invocation in DataGrip
call change_major_proc('CS','CSD');
end;
```

- JAVA:
```
// As of v11, procedures must be outside a transaction to work
conn.setAutoCommit(true);

CallableStatement proc =
             conn.prepareCall("{call change_major_proc(?,?)}");
proc.setObject(1, "CS");
proc.setObject(2, "CSD");
proc.execute();
proc.close();
```

JDBC: Execute/Create

```
// Setup function to call.
Statement stmt = conn.createStatement();
stmt.execute("CREATE FUNCTION refcursorfunc() RETURNS refcursor AS '"
    + " DECLARE "
    + "    mycurs refcursor; "
    + " BEGIN "
    + "    OPEN mycurs FOR SELECT * FROM STUDENT;"
    + "    RETURN mycurs; "
    + " END;' language plpgsql");
stmt.close();

// We must be inside a transaction for cursors to work.
conn.setAutoCommit(false);

// Function call.
CallableStatement func = conn.prepareCall("{? = call refcursorfunc() }");
func.registerOutParameter(1, Types.OTHER);
func.execute();
ResultSet results = (ResultSet) func.getObject(1);
while (results.next()){ // do something with the results.}
results.close();
func.close();
```

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

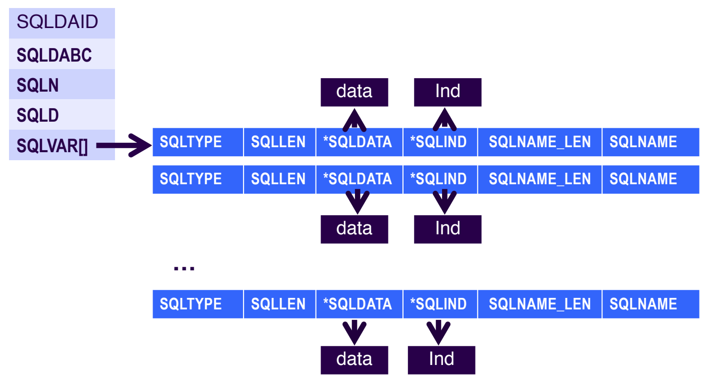
```
DatabaseMetaData dbmd = dbcon.getMetaData();
```

- Native SQL

```
nativeSQL(String sql);
```

  - Converts SQL stmt into the system's native SQL grammar

SQLDA



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

- String sql = "SELECT * FROM user WHERE
          username= '" + username +"' and password='" + password + "'";

- Example inputs:

admin	(for username)
1' OR '1'='1.	(for password)

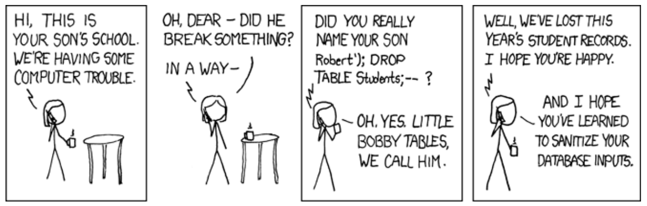
- Result:

```
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 (for username)
  3113'; DELETE FROM user WHERE '1 (for password)
- Result:
  SELECT * FROM user
  WHERE username='panos' and password='3113';
  DELETE FROM user WHERE '1';
- Effect: ?


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

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 = ?");

Fix SQL Injection

```
Connection dbcon =
     DriverManager.getConnection(url, props);
String username = "admin";
String password = "1' OR '1'='1";
PreparedStatement st = dbcon.prepareStatement(
     "SELECT * FROM users WHERE username=? AND password=?");
st.setString(1, username);
st.setString(2, password);
ResultSet rs = st.executeQuery();
if (rs.next()) {
  loggedIn = true;
  System.out.println("Successfully logged in");
} else {
  System.out.println("Username / password not recognized");
}
```