CSc 484
Database Management Systems

Ken Gamradt

Spring 2024

Accessing SQL from a Programming Language

Accessing SQL from a Programming Language

- A database programmer must have access to a general-purpose programming language for at least two reasons
  - Not all queries can be expressed in SQL
    - SQL does not provide the full expressive power of a general-purpose language
  - Non-declarative actions such as
    - printing a report
    - interacting with a user
    - sending the results of a query to a graphical user interface

    cannot be done from within SQL

Accessing SQL from a Programming Language

- There are two approaches to accessing SQL from a general-purpose programming language
  - A general-purpose program
    - Can connect to and communicate with a database server using a collection of functions
  - Embedded SQL
    - Provides a means by which a program can interact with a database server
      - SQL statements are translated at compile time into function calls
      - At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities

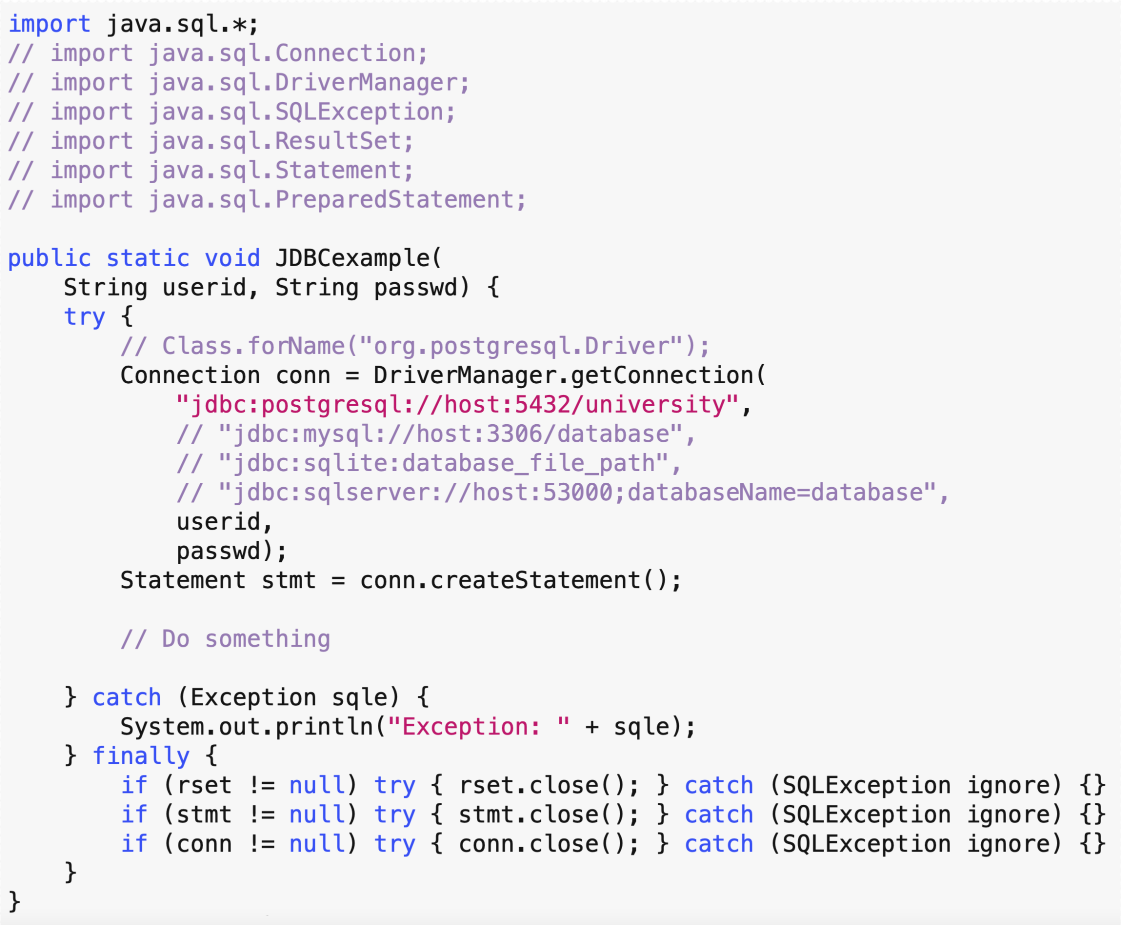
Accessing SQL from a Programming Language

- Dynamic SQL:
  - Allows the program to construct an SQL query as a character string at runtime, submit the query, and then retrieve the result into program variables a tuple at a time
  - String query = “SELECT * FROM instructor;”
- Embedded SQL:
  - The SQL statements are identified at compile time using a preprocessor, which translates requests expressed in embedded SQL into function calls
  - At runtime, the function calls connect to the database
  - EXEC SQL SELECT * FROM instructor;

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL
  - **Java Database Connectivity**
- JDBC supports a variety of features for
  - Querying data
  - Updating data
  - Retrieving query results
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
  - Open a connection
  - Create a "statement" object
  - Execute queries using the statement object to send queries and fetch results
  - Exception mechanism to handle errors

JDBC



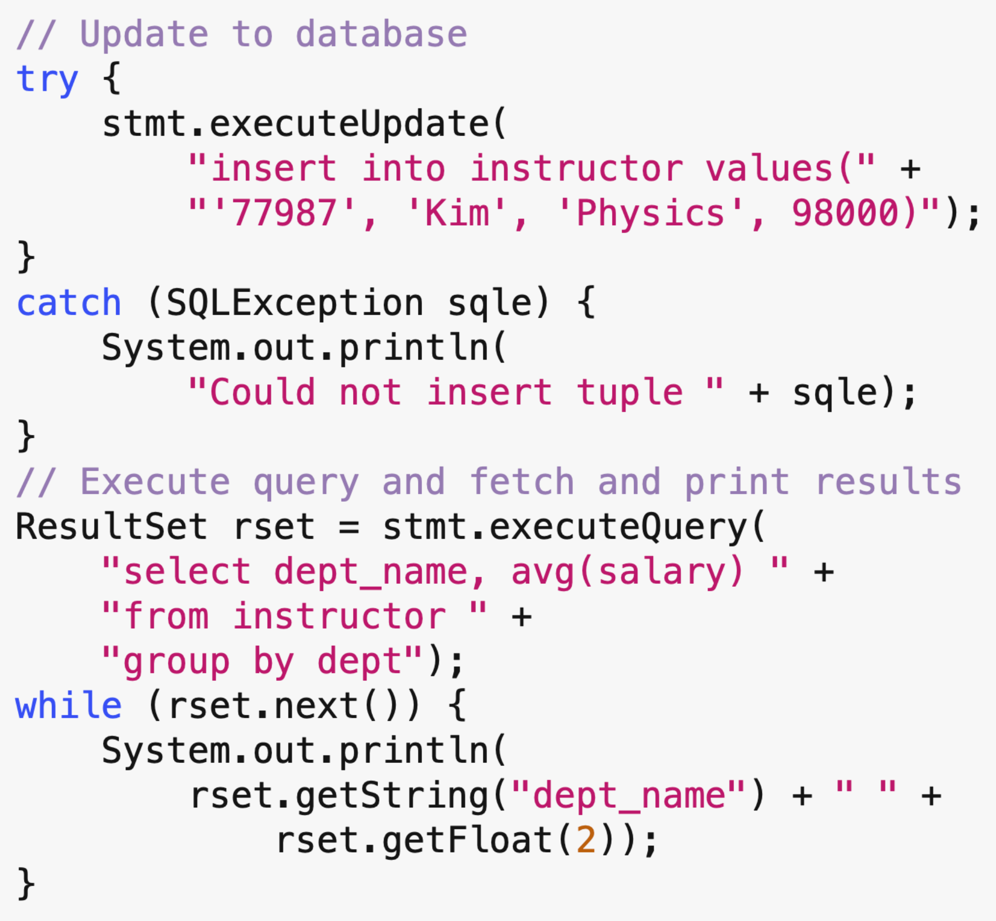
```
import java.sql.*;
// import java.sql.Connection;
// import java.sql.DriverManager;
// import java.sql.SQLException;
// import java.sql.ResultSet;
// import java.sql.Statement;
// import java.sql.PreparedStatement;

public static void JDBCexample(
    String userid, String passwd) {
    try {
        // Class.forName("org.postgresql.Driver");
        Connection conn = DriverManager.getConnection(
            "jdbc:postgresql://host:5432/university",
            // "jdbc:mysql://host:3306/database",
            // "jdbc:sqlite:database_file_path",
            // "jdbc:sqlserver://host:53000;databaseName=database",
            userid,
            passwd);
        Statement stmt = conn.createStatement();

        // Do something

    } catch (Exception sqle) {
        System.out.println("Exception: " + sqle);
    } finally {
        if (rset != null) try { rset.close(); } catch (SQLException ignore) {}
        if (stmt != null) try { stmt.close(); } catch (SQLException ignore) {}
        if (conn != null) try { conn.close(); } catch (SQLException ignore) {}
    }
}
```

JDBC



```
// Update to database
try {
    stmt.executeUpdate(
        "insert into instructor values(" +
        "'77987', 'Kim', 'Physics', 98000)");
}
catch (SQLException sqle) {
    System.out.println(
        "Could not insert tuple " + sqle);
}
// Execute query and fetch and print results
ResultSet rset = stmt.executeQuery(
    "select dept_name, avg(salary) " +
    "from instructor " +
    "group by dept");
while (rset.next()) {
    System.out.println(
        rset.getString("dept_name") + " " +
            rset.getFloat(2));
}
```

JDBC Subsections

- Connecting to the Database
- Shipping SQL Statements to the Database System
- Exceptions and Resource Management
- Retrieving the Result of a Query
- Prepared Statements
- Callable Statements
- Metadata Features
- Other Features
- Database Access from Python

JDBC Code Details

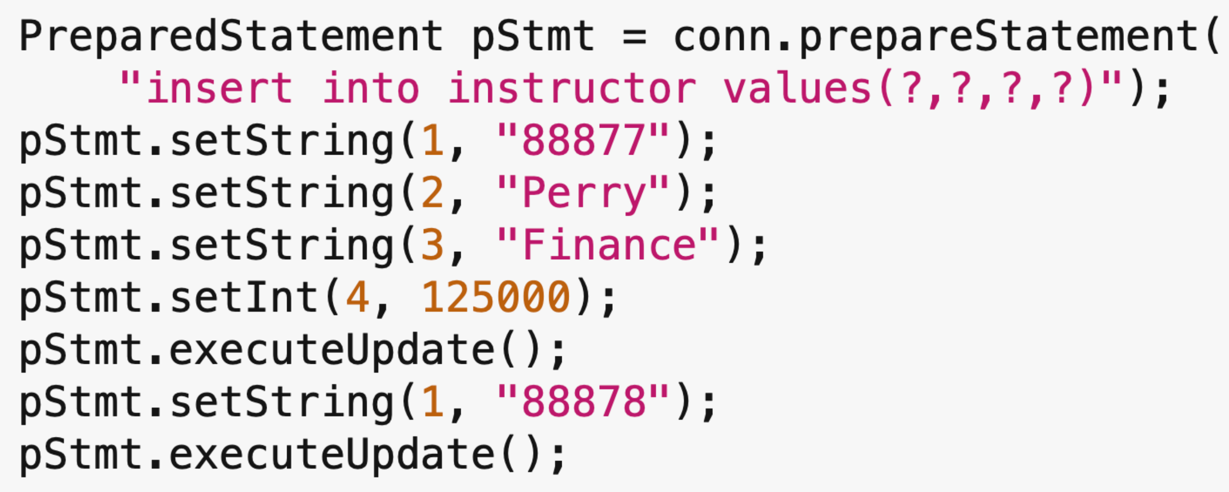
- Getting result fields:

```
// equivalent if dept_name is the first argument of select result
rset.getString(“dept_name”) and rset.getString(1)
```

- Dealing with Null values

```
int a = rset.getInt(“a”);
if (rset.wasNull())
        Systems.out.println(“Got null value”);
```

Prepared Statements



```
PreparedStatement pStmt = conn.prepareStatement(
    "insert into instructor values(?,?,?,?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();
```

- WARNING: always use prepared statements when taking an input from the user and adding it to a query
  - // NEVER create a query by concatenating strings
  - "insert into instructor values(' " + ID + " ', ' " + name + " ', " + " ' " +
    dept name + " ', " ' balance + ')"
  - What if name is “D'Souza”?

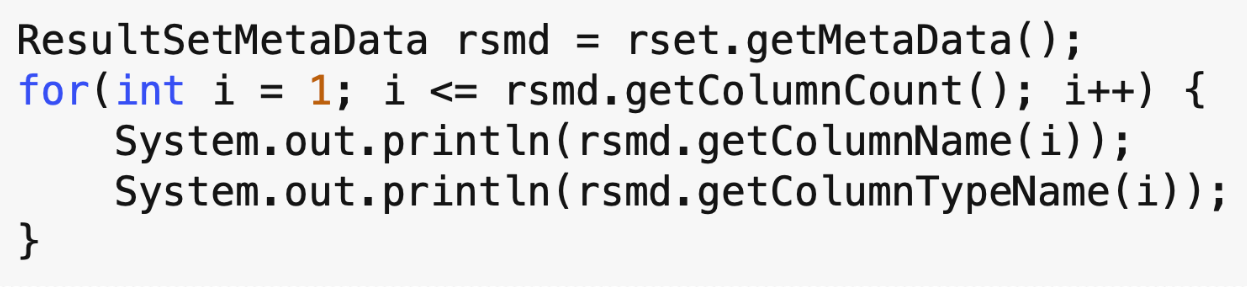
SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = '" + name + "'"
- Suppose the user, instead of entering a name, enters:
  - X' or 'Y' = 'Y
- then the resulting statement becomes:
  - "select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"
  - which is:
    - select * from instructor where name = 'X' or 'Y' = 'Y'
  - User could have even used
    - X'; update instructor set salary = salary + 10000; --
- Prepared statement internally uses:
  - "select * from instructor where name = 'X\' or \'Y\' = \'Y’

- **Note**: Always use prepared statements, with user inputs as parameters

Metadata

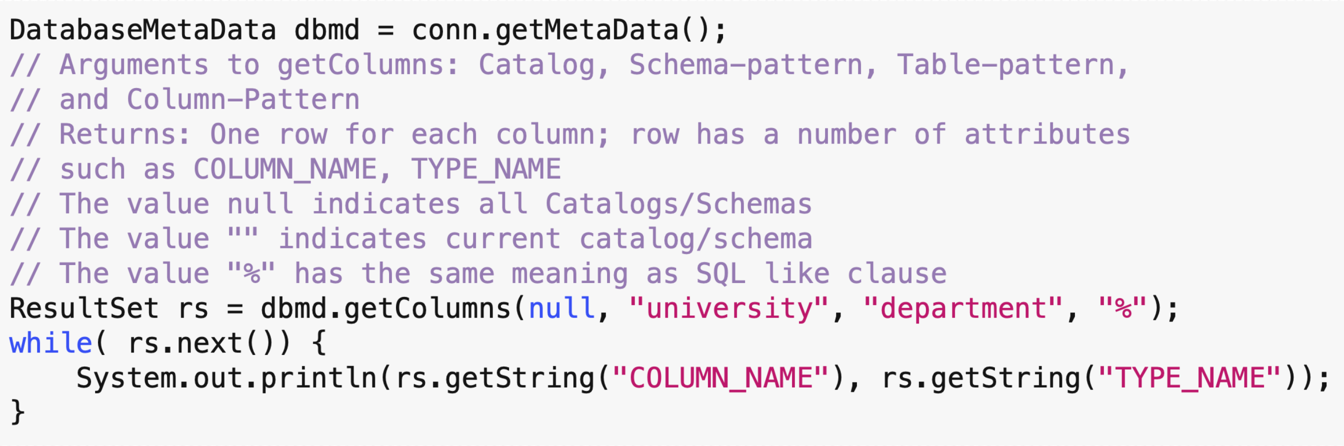
- ResultSet metadata
- E.g., after executing query to get a ResultSet rset:



```
ResultSetMetaData rsmd = rset.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
}
```

Metadata

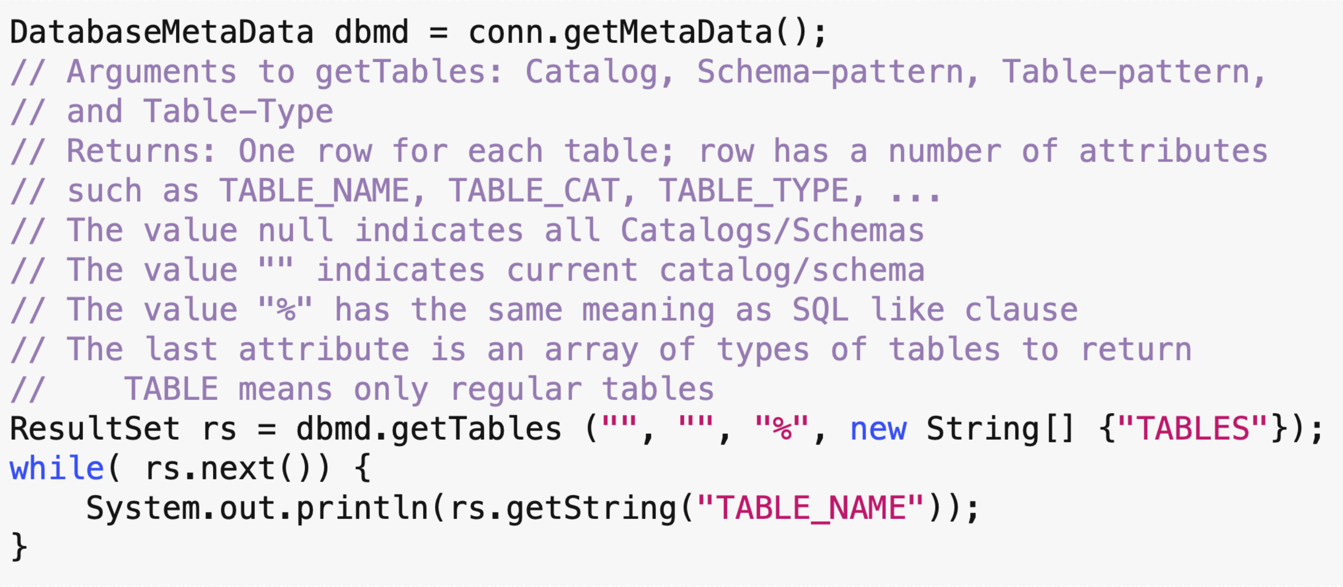
- Database metadata



```
DatabaseMetaData dbmd = conn.getMetaData();
// Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
// and Column-Pattern
// Returns: One row for each column; row has a number of attributes
// such as COLUMN_NAME, TYPE_NAME
// The value null indicates all Catalogs/Schemas
// The value "" indicates current catalog/schema
// The value "%" has the same meaning as SQL like clause
ResultSet rs = dbmd.getColumns(null, "university", "department", "%");
while( rs.next()) {
    System.out.println(rs.getString("COLUMN_NAME"), rs.getString("TYPE_NAME"));
}
```

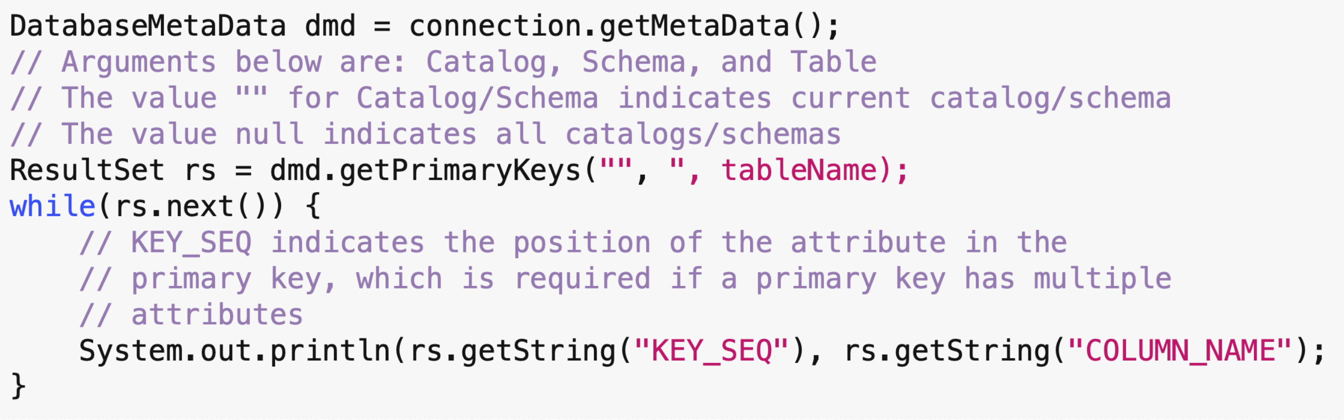
Metadata

- Database metadata



```
DatabaseMetaData dbmd = conn.getMetaData();
// Arguments to getTables: Catalog, Schema-pattern, Table-pattern,
// and Table-Type
// Returns: One row for each table; row has a number of attributes
// such as TABLE_NAME, TABLE_CAT, TABLE_TYPE, ...
// The value null indicates all Catalogs/Schemas
// The value "" indicates current catalog/schema
// The value "%" has the same meaning as SQL like clause
// The last attribute is an array of types of tables to return
//    TABLE means only regular tables
ResultSet rs = dbmd.getTables ("", "", "%", new String[] {"TABLES"});
while( rs.next()) {
    System.out.println(rs.getString("TABLE_NAME"));
}
```

Finding Primary Keys



```
DatabaseMetaData dmd = connection.getMetaData();
// Arguments below are: Catalog, Schema, and Table
// The value "" for Catalog/Schema indicates current catalog/schema
// The value null indicates all catalogs/schemas
ResultSet rs = dmd.getPrimaryKeys("", ", tableName);
while(rs.next()) {
    // KEY_SEQ indicates the position of the attribute in the
    // primary key, which is required if a primary key has multiple
    // attributes
    System.out.println(rs.getString("KEY_SEQ"), rs.getString("COLUMN_NAME");
}
```

Other JDBC Features

- Calling functions and procedures
  - CallableStatement cStmt1 =
    conn.prepareCall("{? = call some function(?)}");
  - CallableStatement cStmt2 =
    conn.prepareCall("{call some procedure(?,?)}");
- Handling large object types
  - getBlob() and getClob() that are similar to the getString() method, but return objects of type Blob and Clob, respectively
  - get data from these objects by getBytes()
  - associate an open stream with Java Blob or Clob object to update large objects
    - blob.setBlob(int parameterIndex, InputStream inputStream)

Other JDBC Features

- A BLOB is binary large object that can hold a variable amount of data with a maximum length of 65535 characters
- These are used to store large amounts of binary data, such as
  - images
  - other types of files
- Fields defined as TEXT also hold large amounts of data
- Difference between the two is the sorts and comparisons done on the stored data
  - case sensitive on BLOBs
  - not case sensitive in TEXT fields
- You do not specify a length with BLOB or TEXT

Other JDBC Features

- CLOB stands for Character Large Object
- SQL Clob is a built-in datatype and is used to store large amount of textual data
  - Can store data up to 2,147,483,647 characters
- The **java.sql.Clob** interface of the JDBC API represents the CLOB datatype
- Since the Clob object in JDBC is implemented using an SQL locator, it holds a logical pointer to the SQL CLOB (not the data)

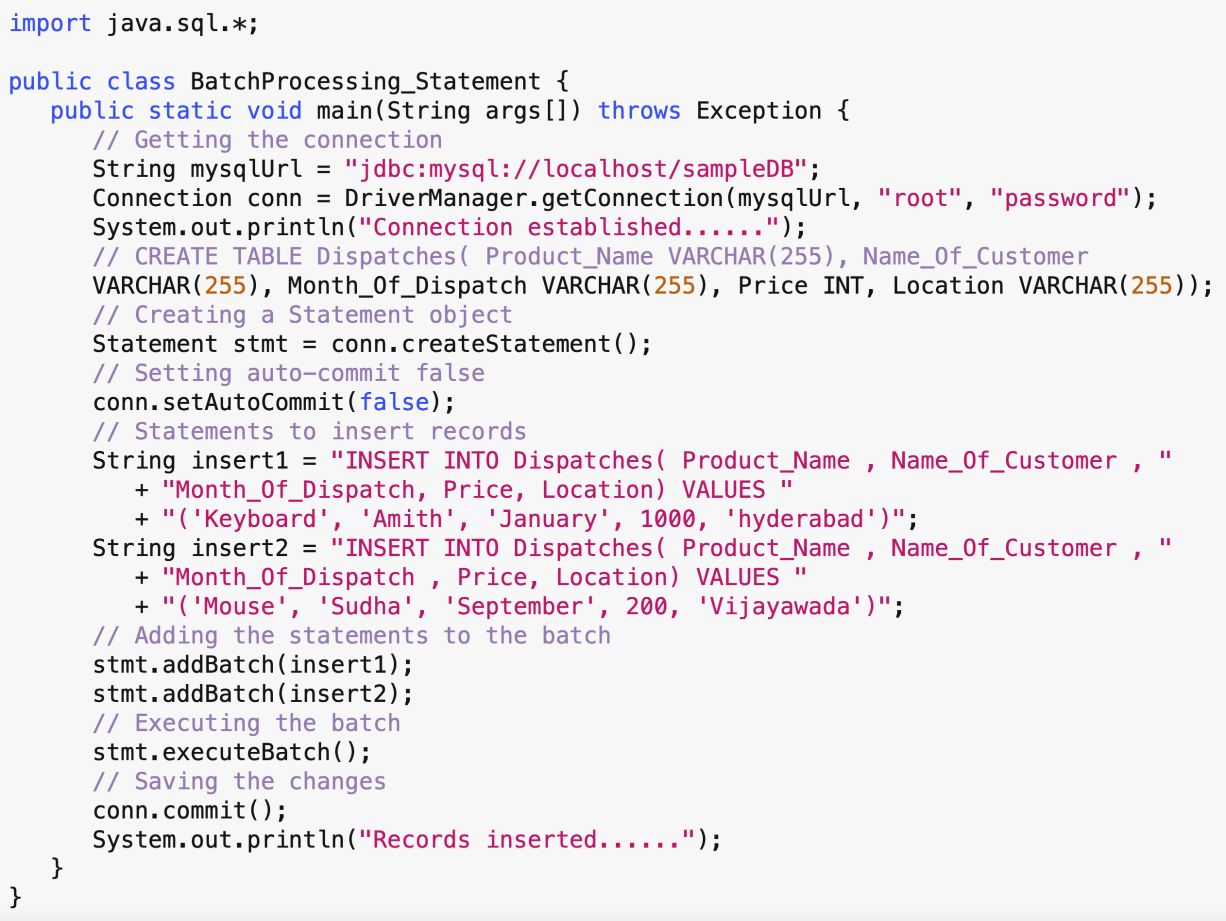
- MYSQL database provides support for this datatype using four variables

TINYTEXT:	maximum of 2^{8}-1 (255) characters
TEXT:	maximum of 2^{16}-1 (65535) characters
MEDIUMTEXT:	maximum of 2^{24}-1 (16777215) characters
LONGTEXT:	maximum of 2^{32}-1 (4294967295) characters

Transaction Control in JDBC

- By default, each SQL statement is treated as a separate transaction that is committed automatically
    - Bad idea for transactions with multiple updates
- Can turn off automatic commit on a connection
    - conn.setAutoCommit(false);
- Transactions must then be committed or rolled back explicitly
    - conn.commit(); or
    - conn.rollback();
- conn.setAutoCommit(true) turns on automatic commit

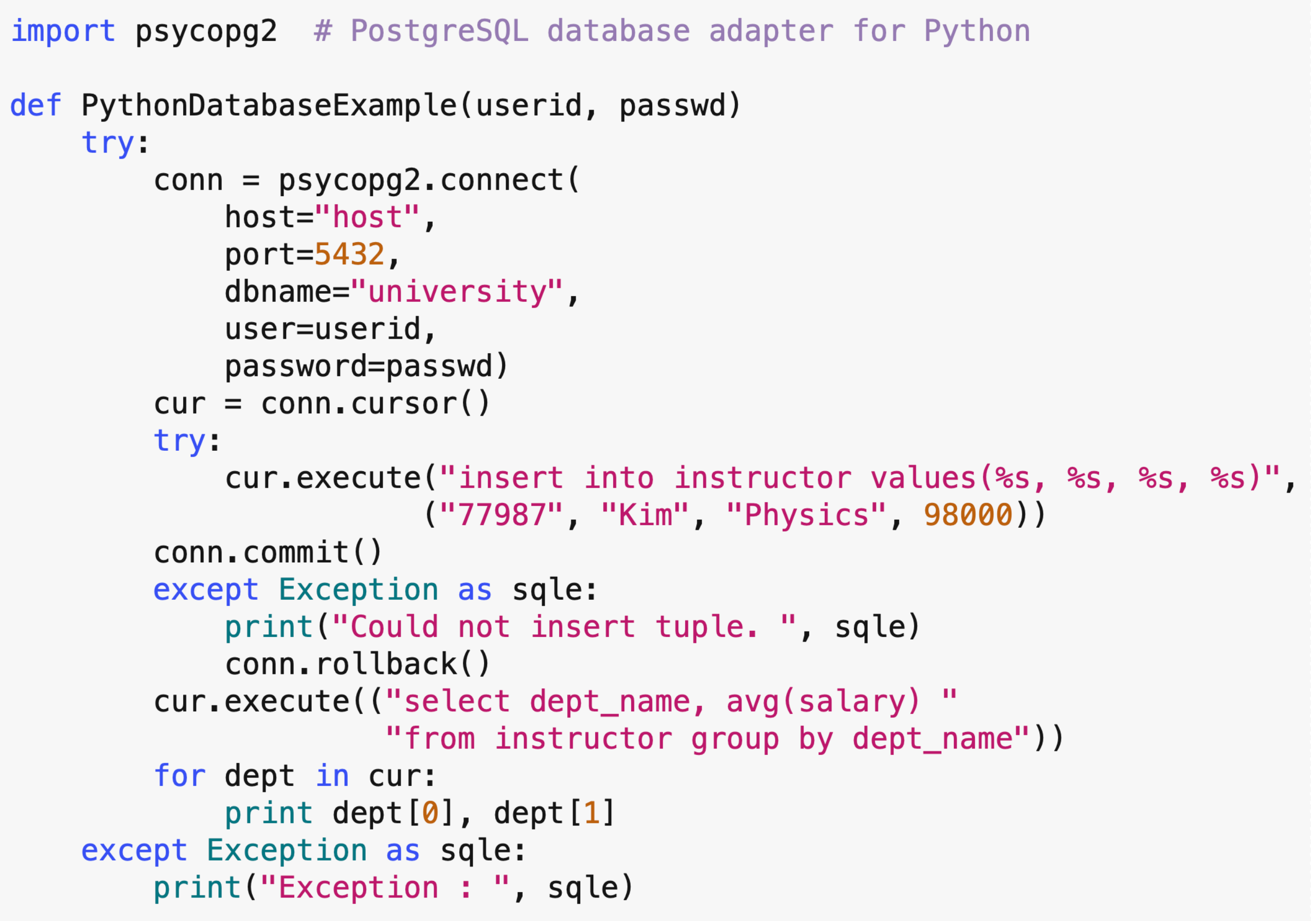
Transaction Control in JDBC



```java
import java.sql.*;

public class BatchProcessing_Statement {
   public static void main(String args[]) throws Exception {
      // Getting the connection
      String mysqlUrl = "jdbc:mysql://localhost/sampleDB";
      Connection conn = DriverManager.getConnection(mysqlUrl, "root", "password");
      System.out.println("Connection established......");
      // CREATE TABLE Dispatches( Product_Name VARCHAR(255), Name_Of_Customer
      VARCHAR(255), Month_Of_Dispatch VARCHAR(255), Price INT, Location VARCHAR(255));
      // Creating a Statement object
      Statement stmt = conn.createStatement();
      // Setting auto-commit false
      conn.setAutoCommit(false);
      // Statements to insert records
      String insert1 = "INSERT INTO Dispatches( Product_Name , Name_Of_Customer , "
         + "Month_Of_Dispatch, Price, Location) VALUES "
         + "('Keyboard', 'Amith', 'January', 1000, 'hyderabad')";
      String insert2 = "INSERT INTO Dispatches( Product_Name , Name_Of_Customer , "
         + "Month_Of_Dispatch , Price, Location) VALUES "
         + "('Mouse', 'Sudha', 'September', 200, 'Vijayawada')";
      // Adding the statements to the batch
      stmt.addBatch(insert1);
      stmt.addBatch(insert2);
      // Executing the batch
      stmt.executeBatch();
      // Saving the changes
      conn.commit();
      System.out.println("Records inserted......");
   }
}
```

Database Access from Python



```
import psycopg2  # PostgreSQL database adapter for Python

def PythonDatabaseExample(userid, passwd)
    try:
        conn = psycopg2.connect(
            host="host",
            port=5432,
            dbname="university",
            user=userid,
            password=passwd)
        cur = conn.cursor()
        try:
            cur.execute("insert into instructor values(%s, %s, %s, %s)",
                       ("77987", "Kim", "Physics", 98000))
        conn.commit()
        except Exception as sqle:
            print("Could not insert tuple. ", sqle)
            conn.rollback()
        cur.execute(("select dept_name, avg(salary) "
                     "from instructor group by dept_name"))
        for dept in cur:
            print dept[0], dept[1]
    except Exception as sqle:
        print("Exception : ", sqle)
```

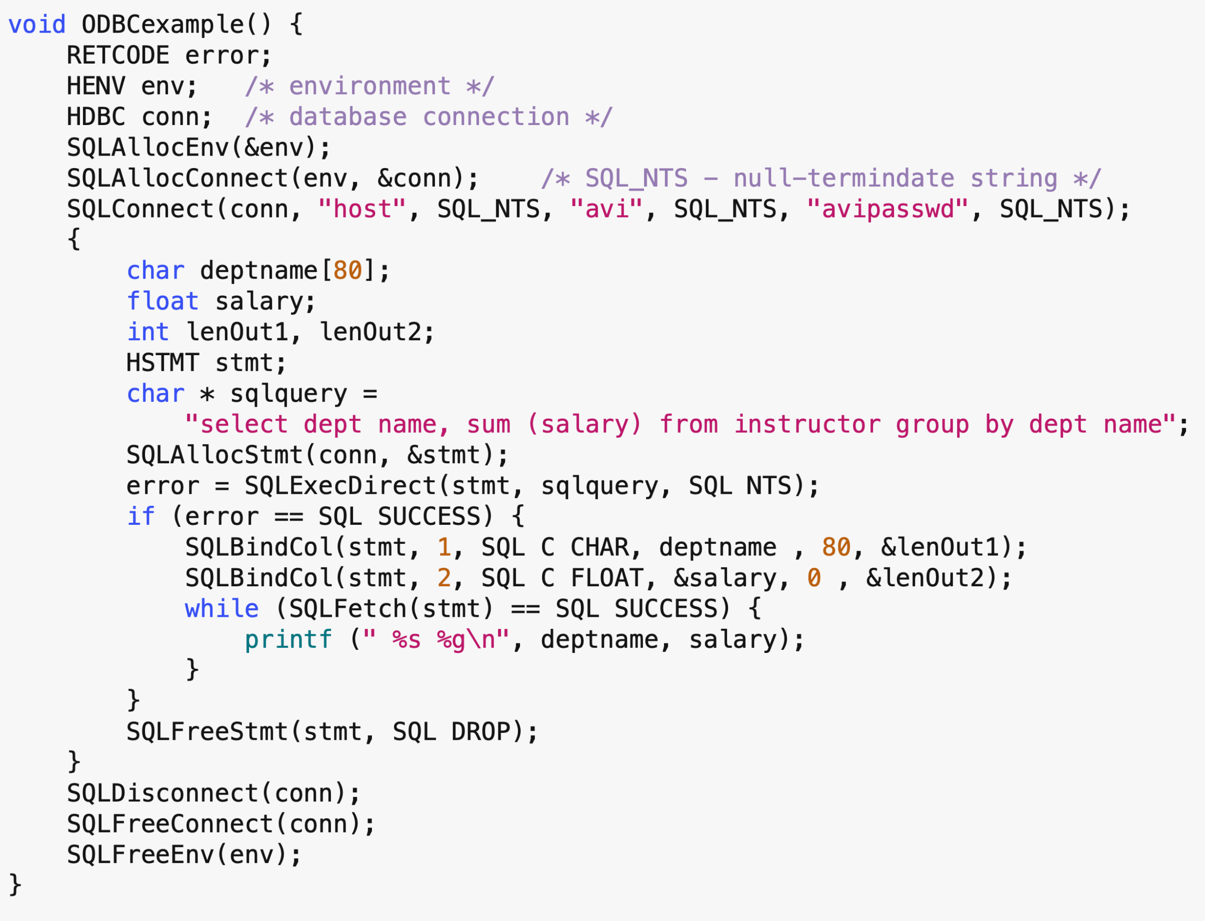
Database Access from Python – The cursor class

- Allows Python code to execute PostgreSQL commands in a database session
- Cursors are created by the connection.cursor() method:
  - they are bound to the connection for the entire lifetime
  - all the commands are executed in the context of the database session wrapped by the connection
- Cursors created from the same connection are not isolated
  - any changes done to the database by a cursor are immediately visible by the other cursors
- Cursors created from different connections can or can not be isolated, depending on the connections' isolation level
  - See also rollback() and commit() methods

ODBC

- **Open DataBase Connectivity** (ODBC) standard
  - standard for application programs to communicate with a database server
  - application program interface (API) to
    - open a connection with a database
    - send queries and updates
    - get back results
- Applications such as GUI, spreadsheets, … can use ODBC

ODBC



```c
void ODBCexample() {
    RETCODE error;
    HENV env;   /* environment */
    HDBC conn;  /* database connection */
    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);    /* SQL_NTS – null–termindate string */
    SQLConnect(conn, "host", SQL_NTS, "avi", SQL_NTS, "avipasswd", SQL_NTS);
    {
        char deptname[80];
        float salary;
        int lenOut1, lenOut2;
        HSTMT stmt;
        char * sqlquery =
            "select dept name, sum (salary) from instructor group by dept name";
        SQLAllocStmt(conn, &stmt);
        error = SQLExecDirect(stmt, sqlquery, SQL NTS);
        if (error == SQL SUCCESS) {
            SQLBindCol(stmt, 1, SQL C CHAR, deptname , 80, &lenOut1);
            SQLBindCol(stmt, 2, SQL C FLOAT, &salary, 0 , &lenOut2);
            while (SQLFetch(stmt) == SQL SUCCESS) {
                printf (" %s %g\n", deptname, salary);
            }
        }
        SQLFreeStmt(stmt, SQL DROP);
    }
    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```

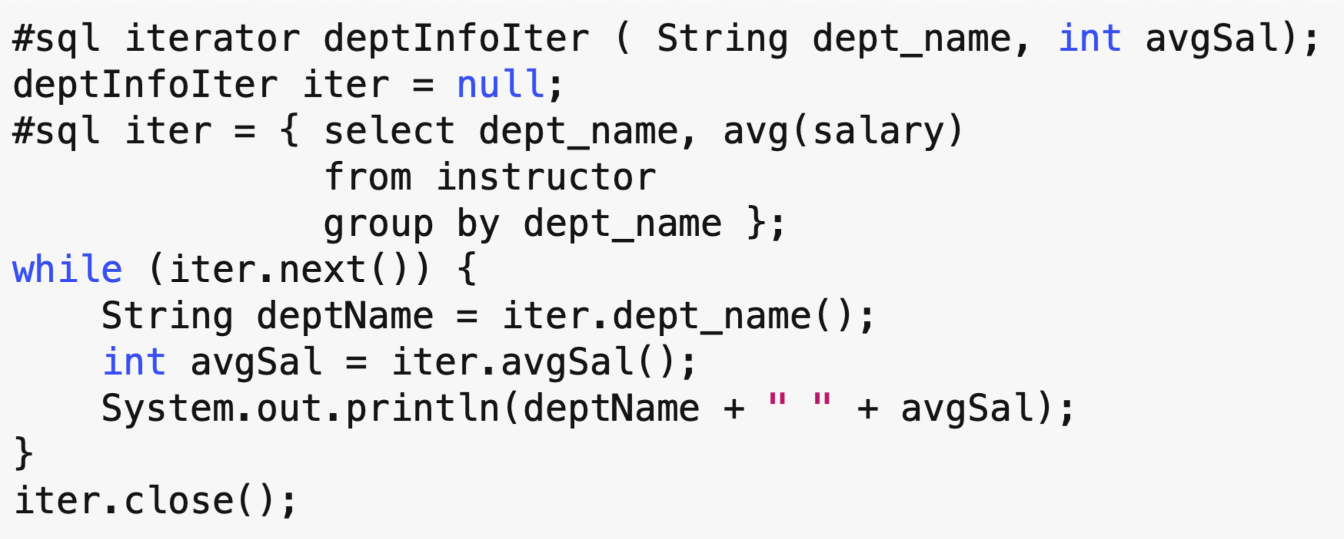
Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise **embedded** SQL
- The basic form of these languages follows that of the System R embedding of SQL into PL/1
- **EXEC SQL** statement is used in the host language to identify embedded SQL request to the preprocessor

  EXEC SQL <embedded SQL statement >;
- **Note**: this varies by language
  - In some languages, like COBOL, the semicolon is replaced with END EXEC
  - In Java embedding uses #SQL { …. };

SQLJ

- JDBC is overly dynamic, errors cannot be caught by compiler
- SQLJ: embedded SQL in Java



```
#sql iterator deptInfoIter ( String dept_name, int avgSal);
deptInfoIter iter = null;
#sql iter = { select dept_name, avg(salary)
              from instructor
              group by dept_name };
while (iter.next()) {
    String deptName = iter.dept_name();
    int avgSal = iter.avgSal();
    System.out.println(deptName + " " + avgSal);
}
iter.close();
```

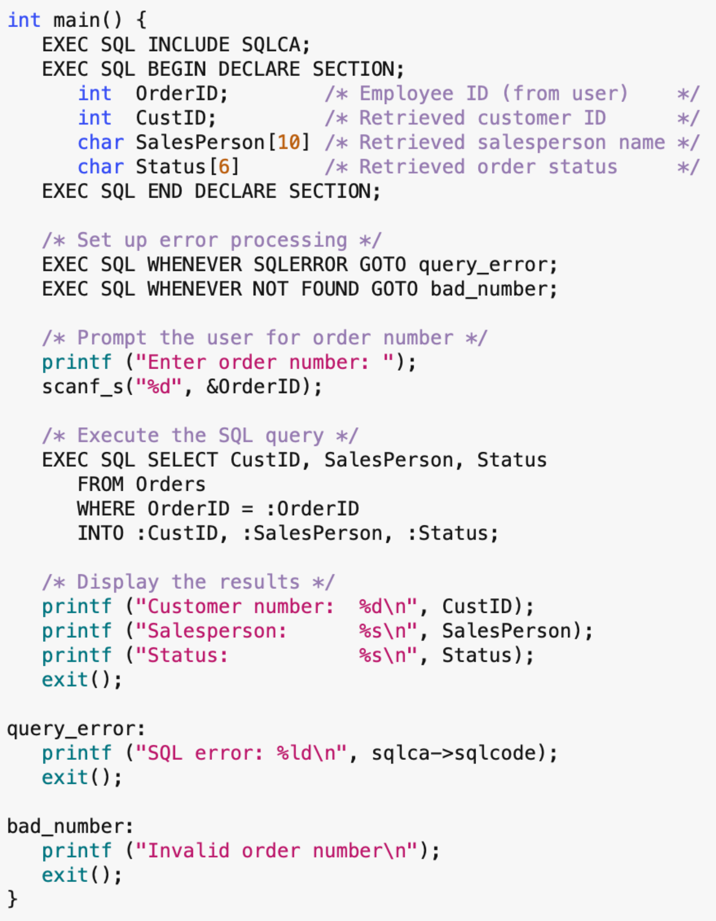
Embedded SQL

- Before executing any SQL statements, the program must first connect to the database

  EXEC SQL **connect to** *server* **user** *user-name* **using** *password*;
- Variables of the host language can be used within embedded SQL statements
  - They are preceded by a colon (:) to them distinguish from SQL variables
    - E.g., *:credit_amount*
- Variables must be declared within the DECLARE section
  - The syntax for declaring the variables follows the host language syntax

```
EXEC SQL BEGIN DECLARE SECTION;
        int  credit_amount ;
EXEC SQL END DECLARE SECTION;
```

Embedded SQL



```
int main() {
   EXEC SQL INCLUDE SQLCA;
   EXEC SQL BEGIN DECLARE SECTION;
      int  OrderID;         /* Employee ID (from user)    */
      int  CustID;          /* Retrieved customer ID      */
      char SalesPerson[10] /* Retrieved salesperson name */
      char Status[6]        /* Retrieved order status     */
   EXEC SQL END DECLARE SECTION;

   /* Set up error processing */
   EXEC SQL WHENEVER SQLERROR GOTO query_error;
   EXEC SQL WHENEVER NOT FOUND GOTO bad_number;

   /* Prompt the user for order number */
   printf ("Enter order number: ");
   scanf_s("%d", &OrderID);

   /* Execute the SQL query */
   EXEC SQL SELECT CustID, SalesPerson, Status
      FROM Orders
      WHERE OrderID = :OrderID
      INTO :CustID, :SalesPerson, :Status;

   /* Display the results */
   printf ("Customer number:  %d\n", CustID);
   printf ("Salesperson:      %s\n", SalesPerson);
   printf ("Status:           %s\n", Status);
   exit();

query_error:
   printf ("SQL error: %ld\n", sqlca->sqlcode);
   exit();

bad_number:
   printf ("Invalid order number\n");
   exit();
}
```

Embedded SQL

- To write an embedded SQL query, use

  declare *c* **cursor for <SQL query>** -- c used to identify the query

- E.g.,
  - From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable **credit_amount** in the host language
  - Specify the query in SQL as follows:

```
EXEC SQL
        declare c cursor for
        select ID, name
        from student
        where tot_cred > :credit_amount
END-EXEC
```

Embedded SQL

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- The **close** statement causes the database system to delete the temporary relation that holds the result of the query

```
EXEC SQL close c ;
```

- **Note**: above details vary with language
  - E.g., Java embedding defines Java iterators to step through result tuples

Updated through Embedded SQL

- Embedded SQL expressions for database modification
  - **update**, **insert**, and **delete**
- Can update tuples fetched by cursor by declaring that the cursor is for update

```
EXEC SQL
            declare c cursor for
            select *
            from instructor
            where dept_name = 'Music'
            for update
```

- Iterate through the tuples by performing **fetch** operations on the cursor, and after fetching each tuple we execute the following code:

```
            update instructor
            set salary = salary + 1000
            where current of c
```

Acknowledgements

- tutorialspoint
  - https://www.tutorialspoint.com/what-is-the-use-of-the-method-setautocommit-in-jdbc
- Psycopg – PostgreSQL database adapter for Python
  - https://www.psycopg.org/docs/cursor.html