# Unit I – Relational Model

Relational Data Model - keys, referential integrity and foreign keys, Relational Algebra - SQL fundamentals- Introduction, data definition in SQL, table, key and foreign key definitions, update behaviors-Intermediate SQL- **Advanced SQL features** -**Embedded SQL- Dynamic SQL,** CASE Studies- Oracle: Database Design and Querying Tools; SQL Variations and Extensions

# 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, since 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, or sending the results of a query to a graphical user interface -- cannot be done from within SQL.

# Accessing SQL from a Programming Language (Cont.)

**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.

  - The 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.

# Java Database Connectivity

- JDBC or Java Database Connectivity is a Java API to connect and execute the query with the database.

- Specification from Sun Microsystems

- *The* **classes and interfaces** *of JDBC* allow the *application to send* requests *made by users to the specified database.*

# Purpose of JDBC

- **Interacting with a database** requires efficient database connectivity, which can be achieved by using the **ODBC(Open database connectivity) driver**

- Driver is used with JDBC to interact or communicate with various kinds of databases such as **Oracle, MS Access, Mysql, and SQL server database**.

1. Java Applications
2. Java Applets
3. Java Servlets
4. Java ServerPages (JSPs)
5. Enterprise JavaBeans (EJBs).

# Architecture
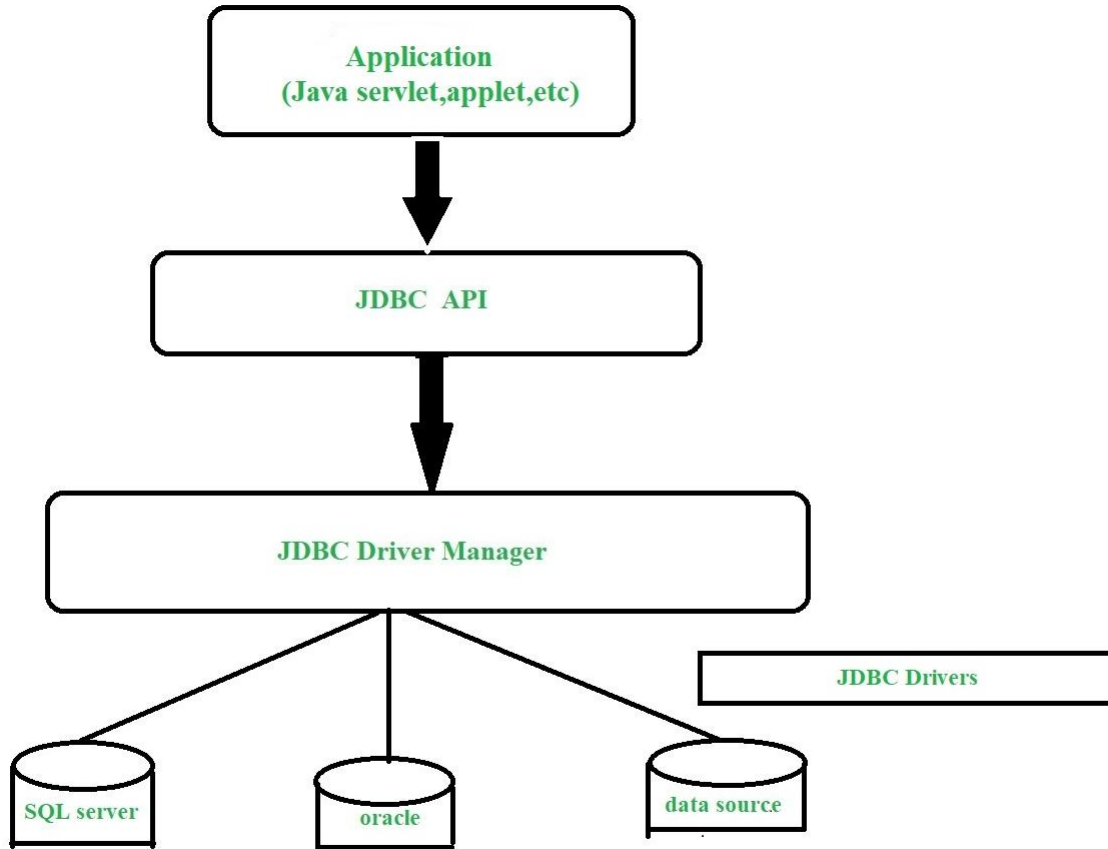
# Components of JDBC

1. **JDBC API:** It provides various methods and interfaces for easy communication with the database.  - **java.sql.*;**

2.  **JDBC Driver manager:** It **loads a database-specific driver in an application** to establish a connection with a database.

3. **JDBC Test suite:** It is used to test the operation(such as insertion, deletion, updation) being performed by JDBC Drivers.

4. **JDBC-ODBC Bridge Drivers**: It connects database drivers to the database. This bridge translates the **JDBC method call to the ODBC function call.**

- **Driver Manager** – This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol.

- **Driver** – This interface handles the communications with the database server.

- **Connection** – This interface with all methods for contacting a database.

- **Statement** – You use objects created from this interface to submit the SQL statements to the database.

- **ResultSet** – These objects hold data retrieved from a database after you execute an SQL query using Statement objects.

- **SQLException -** –This class handles any errors that occur in a database application.
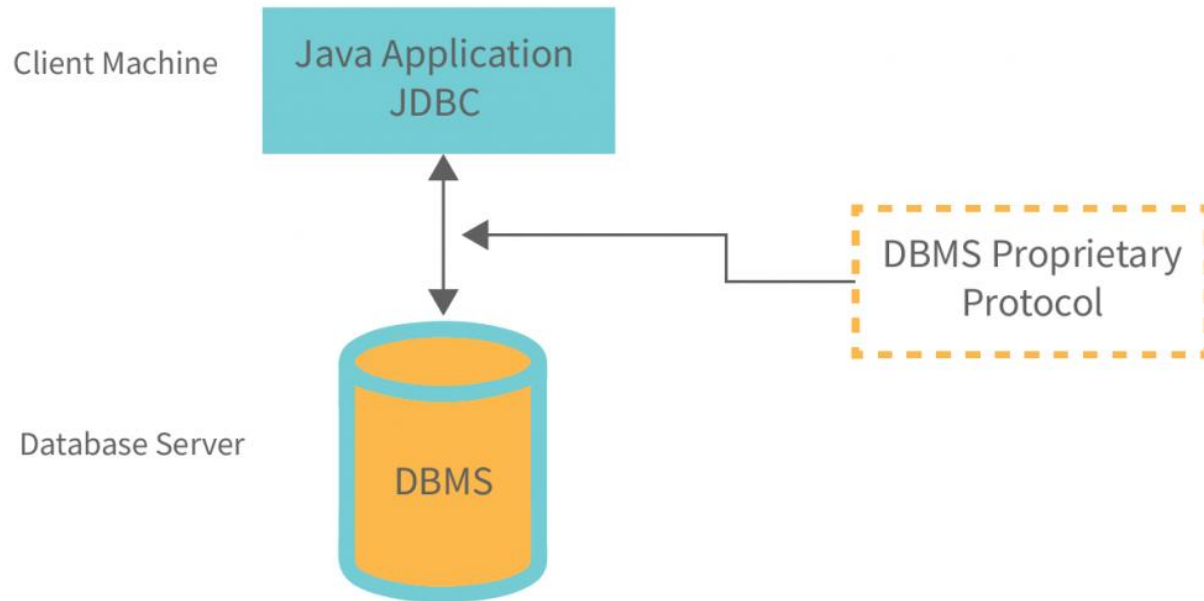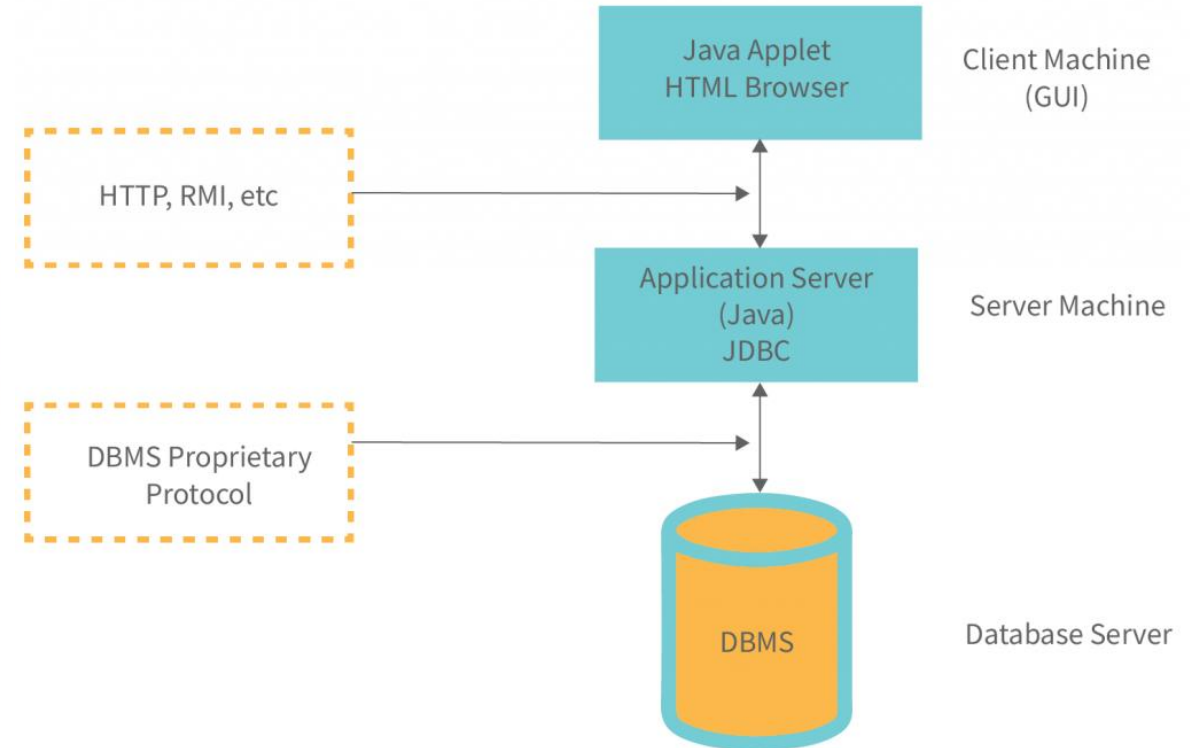
# JDBC Drivers

There are 4 types of JDBC drivers:

1.Type-1 driver or JDBC-ODBC bridge driver

2.Type-2 driver or Native-API driver

3.Type-3 driver or Network Protocol driver

4.Type-4 driver or Thin driver

# Types of JDBC Architecture(2-tier and 3-tier)

# Interfaces of JDBC API

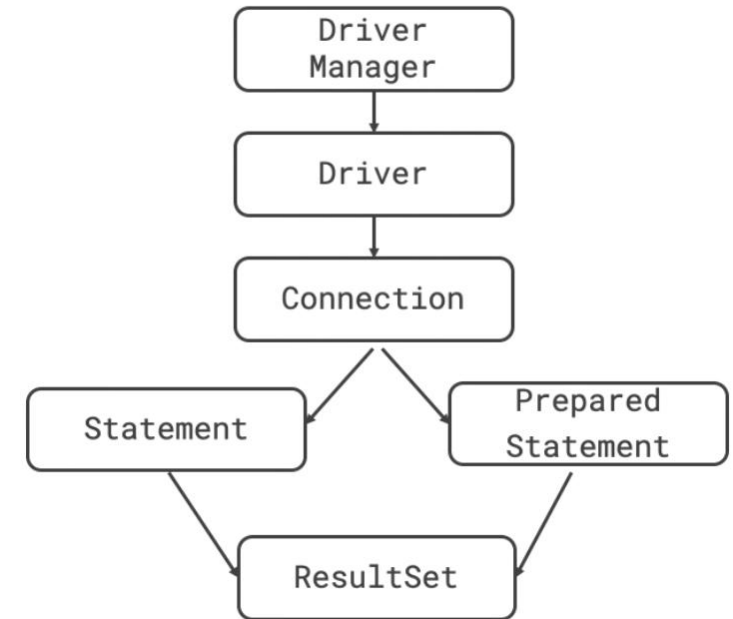- Driver interface

- Connection interface

- Statement interface

- PreparedStatement interface

- CallableStatement interface

- ResultSet interface

- ResultSetMetaData interface

- DatabaseMetaData interface

- RowSet interface

JDBC Interfaces

# **Popular Classes in JDBC API**

Java Database Connectivity

- DriverManager class

- Blob class - Binary

- Clob class - Character

- Types class

Register driver    01

Get connection    02

Create statement    03

Execute query    04

Close connection    05

```java
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
            … Do Actual Work ….
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

**NOTE:  Class.forName is not required from JDBC 4 onwards.**

# Simple Application

```
package com.sa.jdbc;

import java.sql.*;

public class JDBCDemo {

  public static void main(String args[])
    throws SQLException, ClassNotFoundException
  {
    String driverClassName
      = "sun.jdbc.odbc.JdbcOdbcDriver";
    String url = "jdbc:odbc:XE";
    String username = "scott";
    String password = "tiger";
    String query
      = "insert into students values(109, 'bhatt')";

    // Load driver class
    Class.forName(driverClassName);

    // Obtain a connection
    Connection con = DriverManager.getConnection(
      url, username, password);

    // Obtain a statement
    Statement st = con.createStatement();

    // Execute the query
    int count = st.executeUpdate(query);
    System.out.println(
      "number of rows affected by this query= "
      + count);

    // Closing the connection as per the
    // requirement with connection is completed
    con.close();
  }
}
```

# 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_name");
while (rset.next()) {

    System.out.println(rset.getString("dept_name") + " " +  rset.getFloat(2));
}
```

# Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1,

- 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 { .... };**

- Before executing any SQL statements, the program must first connect to the database. This is done using:

**EXEC-SQL connect to  *server*  user *user-name* using *password*;**

Here, *server* identifies the server to which a connection is to be established.

- Variables used as above must be declared within DECLARE section,

**EXEC-SQL BEGIN DECLARE SECTION}**

**int  *credit-amount* ;**

**EXEC-SQL END DECLARE SECTION;**

- To write an embedded SQL query, we use the

  **declare** *c* **cursor for  <SQL query>**

- Example:

  EXEC SQL

  **declare** *c* **cursor for**

  **select** *ID, name*

  **from** *student*

  **where tot_cred** *> :credit_amount*

  END_EXEC

# Embedded SQL (Cont.)

- The **open** statement        EXEC SQL **open** *c* ;

  This statement causes the database system to execute the query and to save the results within a temporary relation. The query uses the value of the host-language variable *credit-amount* at the time the **open** statement is executed.

- The fetch statement causes the values of one tuple in the query result to be placed on host language variables.

  EXEC SQL **fetch** *c* **into** *:si, :sn* END_EXEC

  Repeated calls to fetch get successive tuples in the query result

# Embedded SQL (Cont.)

- 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.  For example, the Java embedding defines Java iterators to step through result tuples.

# Updates 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**

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

# Updates Through Embedded SQL

- Theniterate through the tuples by performing **fetch** operations on the cursor (as illustrated earlier), and after fetching each tuple we execute the following code:

# Difference between Embedded and Dynamic SQL

- **Static or Embedded** SQL are SQL statements in an application that do not change at runtime and, therefore, can be hard-coded into the application.

- **Dynamic** SQL is SQL statements that are constructed at runtime; for example, the application may allow users to enter their own queries.

# **Functions and Procedure**

- The function program has a block of code that performs some specific tasks or functions.

- Particular set of instructions or commands along known as a procedure.

CREATE [OR REPLACE] FUNCTION function_name

[(parameter_name  type [, ...])]

// this statement  is must for functions

RETURN return_datatype

{IS | AS}

BEGIN

   // program code

[EXCEPTION

 exception_section;

END [function_name];

**create function MultiplyNumbers(@int1 as int,@int2 as int)**
**As**
**BEGIN**
**Return (@int1 * @int2)**
**end**

# Procedure

CREATE or REPLACE PROCEDURE name(parameters)
IS
variables;
BEGIN
//statements;
END;

CREATE or REPLACE PROCEDURE INC_SAL(eno IN NUMBER, up_sal OUT NUMBER)
IS
BEGIN
UPDATE emp_table SET salary = salary+1000 WHERE emp_no = eno;
COMMIT;
SELECT sal INTO up_sal FROM emp_table WHERE emp_no = eno;
END;

# Trigger

- A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs.

- For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

create trigger [trigger_name]

[before | after]

{insert | update | delete}

on [table_name]

[for each row]

[trigger_body]

# Trigger Example

```
mysql> desc Student;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| tid   | int(4)      | NO   | PRI | NULL    | auto_increment |
| name  | varchar(30) | YES  |     | NULL    |                |
| subj1 | int(2)      | YES  |     | NULL    |                |
| subj2 | int(2)      | YES  |     | NULL    |                |
| subj3 | int(2)      | YES  |     | NULL    |                |
| total | int(3)      | YES  |     | NULL    |                |
| per   | int(3)      | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
7 rows in set (0.00 sec)
```

# Trigger Example

```
create trigger stud_marks

before

INSERT

on Student

for each row

set Student.total = Student.subj1 + Student.subj2 +

Student.subj3, Student.per = Student.total * 60 / 100;
```

# Trigger Example

```
mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);
Query OK, 1 row affected (0.09 sec)

mysql> select * from Student;
+------+-------+-------+-------+-------+-------+------+
| tid  | name  | subj1 | subj2 | subj3 | total | per  |
+------+-------+-------+-------+-------+-------+------+
| 100  | ABCDE |    20 |    20 |    20 |    60 |   36 |
+------+-------+-------+-------+-------+-------+------+
1 row in set (0.00 sec)
```

Thank You!