GUJARAT TECHNOLOGICAL UNIVERSITY

MASTER OF COMPUTER APPLICATIONS

SEMESTER: III

Subject : **4639302**

Programming in JAVA

UNIT – 6 Database Programming

The Design of JDBC, JDBC Driver Types, SQL, JDBC Configuration: URL, driver jar files, starting the database, registering the driver class, connecting to the database, Working with JDBC Statements: executing SQL statement, managing connections, statements, resultsets, SQL exceptions, Query Execution: prepared statement.

The JDBC API

- Java Database Connectivity is an API for interacting with a database.
- Each database vendor produces a *driver* that translates JDBC requests to the database-specific protocol.
- JDBC is based on the Standard Query Language for database access.
- JDBC provides classes for issuing SQL statements and for working with the query results.
- JDBC has been hugely successful.
  - Every commonly used database has a JDBC driver.
  - There are several databases written in Java.
- Object-relational mappers provide a higher-level mechanism for working with a database, but you should still understand SQL/JDBC to use them effectively.

BipinRupadiya.com

The Design of JDBC

- ➢ From the start, the developers of the Java technology were aware of the potential that Java showed for working with databases.
- ➢ In 1995, they began working on extending the standard Java library to deal with SQL access to databases.
- ➢ What they first hoped to do was to extend Java so that a program could talk to any random database using only “pure” Java.
- ➢ It didn’t take them long to realize that this is an impossible task:
  - ❑ There are simply too many databases out there, using too many protocols.
  - ❑ Moreover, although database vendors were all in favor of Java providing a standard network protocol for database access, they were only in favor of it if Java used *their* network protocol.

BipinRupadiya.com

The Design of JDBC

- What all the database vendors and tool vendors *did* agree on was that it would be useful for Java to provide a pure Java API for SQL access along with a driver manager to allow third-party drivers to connect to specific databases.
- Database vendors could provide their own drivers to plug in to the driver manager.
- There would then be a simple mechanism for registering third-party drivers with the driver manager.

BipinRupadiya.com

The Design of JDBC

- This organization follows the very successful model of Microsoft’s ODBC which provided a C programming language interface for database access.
- Both JDBC and ODBC are based on the same idea:
  - ❑ Programs written according to the API talk to the driver manager, which, in turn, uses a driver to talk to the actual database.
- This means the JDBC API is all that most programmers will ever have to deal with.

BipinRupadiya.com

JDBC Driver Types

- **The JDBC specification classifies drivers into the Four *types*:**
- **Type-1 (*JDBC/ODBC bridge*)**
  - *type 1 driver* translates JDBC to ODBC and relies on an ODBC driver to communicate with the database.
  - Early versions of Java included one such driver, the *JDBC/ODBC bridge*.
  - However, the bridge requires deployment and proper configuration of an ODBC driver.
  - When JDBC was first released, the bridge was handy for testing, but it was never intended for production use.
  - At this point, many better drivers are available, and Java 8 no longer provides the JDBC/ODBC bridge.

BipinRupadiya.com

JDBC Driver Types

- ***Type 2*** **(partly in Java *)***
  - *type 2 driver* is written partly in Java and partly in native code;
  - it communicates with the client API of a database.
  - When using such a driver, you must install some platform-specific code onto the client in addition to a Java library.
- ***Type 3*** **(pure Java client library*)***
  - *type 3 driver* is a pure Java client library that uses a database-independent protocol to communicate database requests to a server component, which then translates the requests into a database-specific protocol.
  - This simplifies deployment because the platform-specific code is located only on the server.
- ***Type 4*** **(pure Java library*)***
  - *type 4 driver* is a pure Java library that translates JDBC requests directly to a database-specific protocol.

BipinRupadiya.com

JDBC Driver Types

- Most database vendors supply either a type 3 or type 4 driver with their database.
- Furthermore, a number of third-party companies specialize in producing drivers with better standards conformance, support for more platforms, better performance, or, in some cases, simply better reliability than the drivers provided by the database vendors.

BipinRupadiya.com

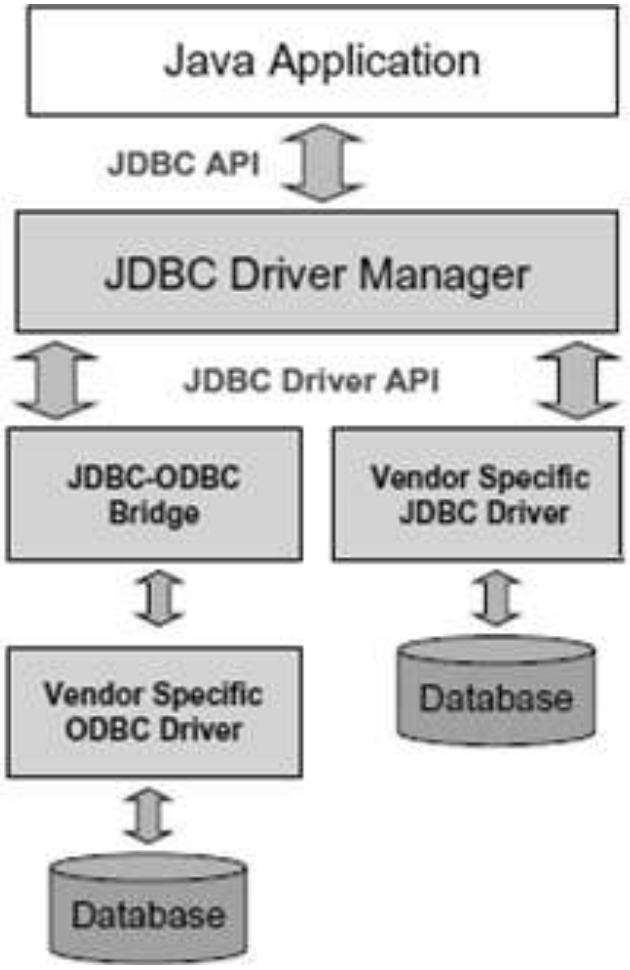
JDBC

- **JDBC provides a standard library for accessing relational databases**
  - API standardizes
    - Way to establish connection to database
    - Approach to initiating queries
    - Method to create stored (parameterized) queries
    - The data structure of query result (table)
      - Determining the number of columns
      - Looking up metadata, etc.
  - API does not standardize SQL syntax
    - JDBC is not embedded SQL
- JDBC classes are in the java.sql package

BipinRupadiya.com

JDBC Drivers consists of two parts

- JDBC is purely Java-based API
- JDBC Driver Manager, which communicates with vendor-specific drivers that perform the real communication with the database.
  - Translation to vendor format is performed on the client
    - No changes needed to server
    - Driver (translator) needed on client



BipinRupadiya.com

Why
Java didn't adopt the ODBC model ?

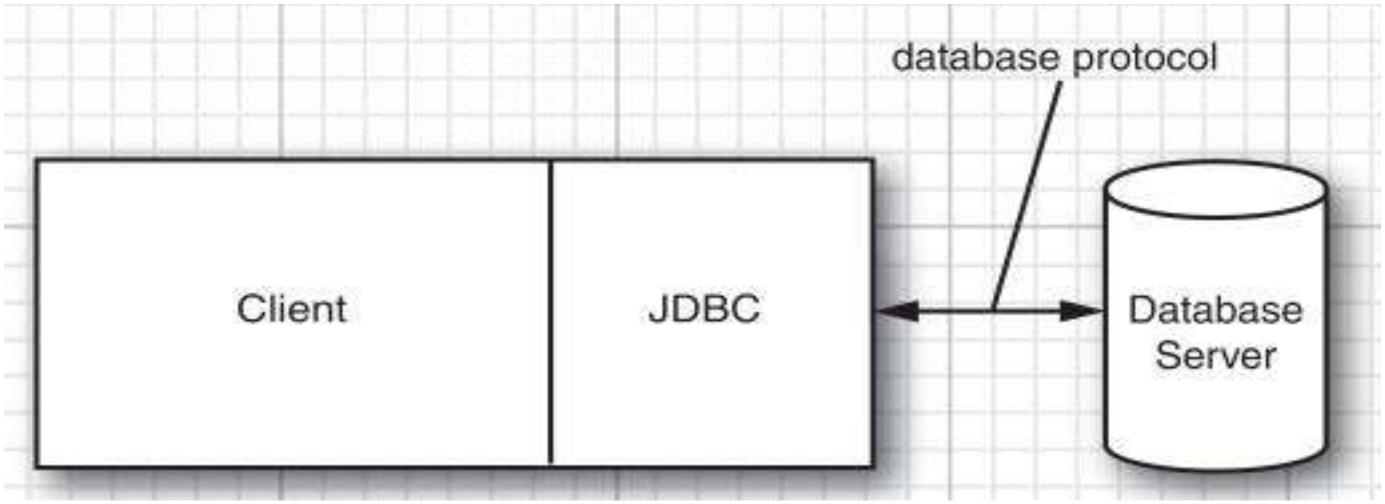
JDBC vs ODBC

- ODBC is hard to learn.
- ODBC has a few commands with lots of complex options. The preferred style in the Java programming language is to have simple and intuitive methods, but to have lots of them.
- ODBC relies on the use of void* pointers and other C features that are not natural in the Java programming language.
- An ODBC-based solution is inherently less safe and harder to deploy than a pure Java solution.

BipinRupadiya.com

Typical Uses of JDBC

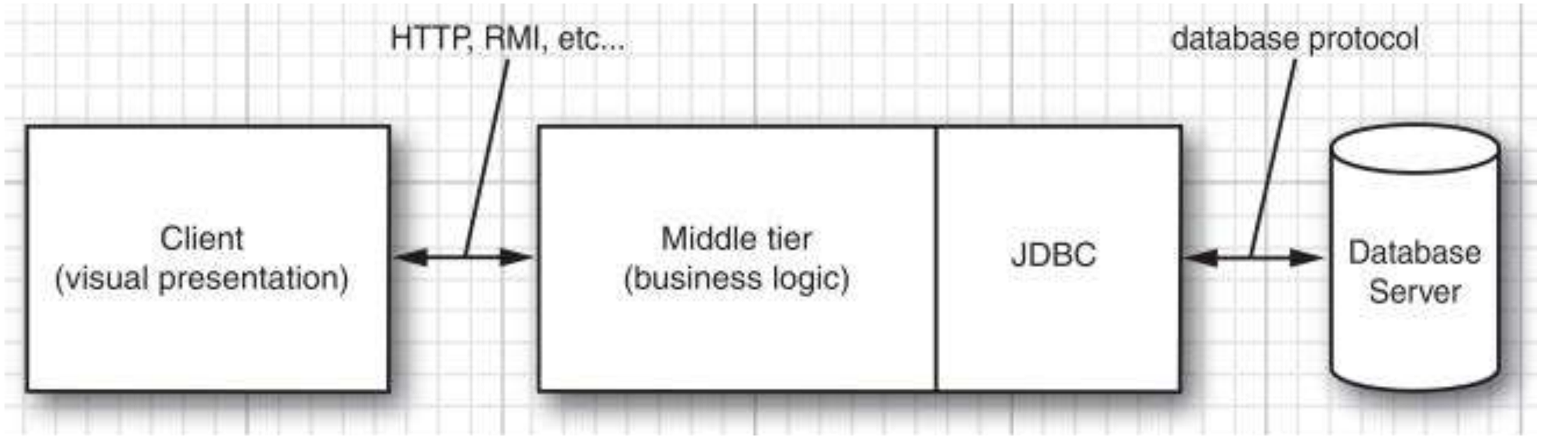
- A traditional client/server application
  - The traditional client/server model has a rich GUI on the client and a database on the server.
  - In this model, a JDBC driver is deployed on the client.



BipinRupadiya.com

Typical Uses of JDBC

- **A three-tier application**
  - it calls on a middleware layer on the server that in turn makes the database queries.
  - The three-tier model has a couple of advantages.
  - It separates *visual presentation* (on the client) from the *business logic* (in the middle tier) and the raw data (in the database).
  - Therefore, it becomes possible to access the same data and the same business rules from multiple clients, such as a Java desktop application, a web browser, or a mobile app.



BipinRupadiya.com

Common SQL Data Types

- **INTEGER or INT** Typically, a 32-bit integer
- **SMALLINT** Typically, a 16-bit integer
- *Fixed-point decimal number with m total digits and n digits after the decimal point*
  - **NUMERIC(m,n), DECIMAL(m,n) or DEC(m,n) FLOAT(n)**
- *A floating-point number with n binary digits of precision*
  - **REAL** Typically, a 32-bit floating-point number
  - **DOUBLE** Typically, a 64-bit floating-point number
- **CHARACTER(n) or CHAR(n)** Fixed-length string of length *n*
- **VARCHAR(n)** Variable-length strings of maximum length n
- **BOOLEAN** A Boolean value
- **DATE** Calendar date, implementation-dependent
- **TIME** Time of day, implementation-dependent
- **TIMESTAMP** Date and time of day, implementation-dependent
- **BLOB** A binary large object
- **CLOB** A character large object

BipinRupadiya.com

Using JDBC

1. Load the driver
2. Define the Connection URL
3. Establish the Connection
4. Create a Statement object
5. Execute a query
6. Process the results
7. Close the connection

BipinRupadiya.com

1. Load the driver

- **Not required in Java 6 or above**
  - JDBC 4.0 and later, the driver is loaded automatically
- **Java 5 and earlier**
  - Load the driver *class* only. The class has a static initialization block that makes an instance and registers it with the **DriverManager.**

```
try {
                //ORACEL
                DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
                Class.forName("oracle.jdbc.driver.OracleDriver");
                //MySQL
                DriverManager.registerDriver(new com.mysql.jdbc.Driver());
                Class.forName("com.mysql.jdbc.Driver")
}
catch(ClassNotFoundException cnfe) {
                System.out.println("Error loading driver: " cnfe);        }
```

BipinRupadiya.com

2. Define the Connection URL

String host = "dbhost.yourcompany.com";

String dbName = "someName";

int port = 1234;

String msAccessURL = "**jdbc:odbc:**" + **dbName**;

// String oracleURL = "**jdbc:oracle:thin:@**" + **host** + ":" + **port** + ":" + **dbName**;

String oracleURL = "jdbc:oracle:thin:@localhost:1521:xe ";

// String mySqlUrl = "**jdbc:mysql//**" + **host** + ":" + **port** +"/" + **dbName**;

String mySqlUrl = "jdbc:mysql://localhost:3306/wtad" ;

// String derbyurl = **"jdbc:derby://**" + **host** + ":" + **port** +"/" + **dbName+";**create=true";

String derbyurl = **"** jdbc:derby://localhost:1527/COREJAVA;create=true "**;**

BipinRupadiya.com

3. Establish the Connection

String username = "bipin";

String password = "rupadiya";

Connection connection =**DriverManager.getConnection**

(oracleURL, username, password);

- **Optionally, look up information about the database**

  DatabaseMetaData dbMetaData =**connection.getMetaData();**

  String productName =**dbMetaData.getDatabaseProductName();**

  System.out.println("Database: " + productName);

  String productVersion =**dbMetaData.getDatabaseProductVersion();**

  System.out.println("Version: " + productVersion);

4. Create a Statement

- A Statement is used to send queries or commands

- **Types**
  - Statement,
  - PreparedStatement,
  - CallableStatement

- **Example**
  - **Statement statement =connection.createStatement();**

BipinRupadiya.com

5. Execute a Query

- To modify the database, use executeUpdate, supplying a string that uses UPDATE, INSERT, or DELETE otherwise use simple executeQuery();
  - ❑ statement.executeQuery("SELECT … FROM …");
    - This version returns a ResultSet
  - ❑ statement.executeUpdate("UPDATE …");
  - ❑ statement.executeUpdate("INSERT …");
  - ❑ statement.executeUpdate("DELETE…");
  - ❑ statement.execute("CREATE TABLE…");
  - ❑ statement.execute("DROP TABLE …");
- **Example**
  - ❑ **String query ="SELECT col1, col2, col3 FROM sometable";**
  - ❑ **ResultSet resultSet=statement.executeQuery(query);**

BipinRupadiya.com

6. Process the Result

- **Important ResultSet methods**
  - **resultSet.next()**
    - Goes to the next row. Returns false if no next row.
  - **resultSet.getString("columnName")**
    - Returns value of column with designated name in current row, as a String. Also getInt, getDouble, getBlob, etc.
  - **resultSet.getString(columnIndex)**
    - Returns value of designated column. First index is 1 not 0
  - **resultSet.beforeFirst()**
    - Moves cursor before first row, as it was initially. Also first
  - **resultSet.absolute(rowNum)**
    - Moves cursor to given row (starting with 1). Also last & afterLast.

BipinRupadiya.com

6. Process the Result

- **Assumption**
  - Query was “SELECT first, last, address FROM…”
- **Using column names**

```
while(resultSet.next())  {
        System.out.printf("First name: %s, last name: %s, address: %s \n",
                                                        resultSet.getString("first"),
                                                        resultSet.getString("last"),
                                                        resultSet.getString("address"));
}
```

- **Using column indices**

```
while(resultSet.next()) {
        System.out.printf("First name: %s, last name: %s, address: %s \n" ,
                                                        resultSet.getString(1),
                                                        resultSet.getString(2),
                                                        resultSet.getString(3));
}
```

7. Close the Connection

```
connection.close();
```

- Since opening a connection is expensive, postpone this step if additional database operations are expected

BipinRupadiya.com

Install Derby

1. **DOWNLOAD**

   https://db.apache.org/derby/derby_downloads.html
2. **EXTRACT Zip**

   db-derby-10.14.2.0-bin.zip
3. **COPY Director and Move to proper place**

   C:\Program Files\Java\db-derby-10.14.2.0-bin
4. **CREATE or Edit Environment Variable**
   - ❑ **DERBY_HOME :**
     - C:\Program Files\Java\db-derby-10.14.2.0-bin\;
   - ❑ **PATH :**
     - C:\Program Files\Java\db-derby-10.14.2.0-bin\bin;
   - ❑ **CLASSPATH :**
     - C:\Program Files\Java\db-derby-10.14.2.0-bin\lib\derbyclient.jar;

BipinRupadiya.com

Requirements for Using JDBC

- You need a database.
  - In this lesson, we use Apache Derby.
  - You can also use MySQL, PostgreSQL, Oracle, Microsoft SQL Server, and so on.
  - Go ahead and start the database now. To start Derby, run:
  - java -jar derby/lib/derbyrun.jar server start
- You need the database driver JAR,
  - such as derbyclient.jar for Derby.
- You need the format of the JDBC URL. For Derby, it is:
  - jdbc:derby://localhost:1527/databaseName;create=true

BipinRupadiya.com

Connecting to the Database

- You need to have the database driver JAR on the class path.
- Use this code to establish a JDBC connection:

```
String url = "jdbc:derby://localhost:1527/COREJAVA;create=true";
String username = "app";
String password = "secret";
Connection conn = DriverManager.getConnection(url, username, password);
```

- You use the Connection object to create SQL statements.
- Database connections are a limited resource. When you are done, be sure to close the connection.

Executing SQL Statements

- **To execute a SQL statement, create a Statement object:**
  - Statement stat = conn.createStatement();
- **To update the database (INSERT, UPDATE, DELETE), call the executeUpdate method:**
  - String command = "UPDATE Books SET Price = Price - 5.00 WHERE Title NOT LIKE '%Introduction%'";
  - **int rows = stat.executeUpdate(command);**
  - The number of affected rows is returned.
- **Use executeQuery to issue a SELECT query:**
  - ResultSet result = stat.executeQuery("SELECT * FROM Books");
- **Use execute to issue arbitrary SQL commands.**

BipinRupadiya.com

Managing JDBC Objects

- **A Connection object can produce one or more Statement objects.**
  - Some database drivers only allow one Statement at a time.
  - Call **DatabaseMetaData.getMaxStatements()** to find out.
- **You can use the same Statement object for multiple queries.**
- **A Statement can have at most one open ResultSet.**
  - Don't work with multiple result sets at a time.
  - Issue a query that gives you all data in one result set.
- **When you execute another query or close a Statement, an open result set is closed.**
  - The **Statement.closeOnCompletion()** method closes the statement as soon as an open result set is closed.
- **When you close a Connection, all statements are closed.**

BipinRupadiya.com

SQL Exceptions and Warnings

- **A SQLException has a chain of SQLException objects.**
- **This is in addition to the “cause” chain that all exceptions have.**
- The **SQLException** class **extends** the **Iterable<Throwable>** interface.
- **You can iterate over all exceptions like this:**

```
for (Throwable t : sqlException) {
        do something with t
}
```

- **There is also a chain of warnings:**

```
SQLWarning w = stat.getWarning();
while (w != null) {
            do something with w
            w = w.nextWarning();
}
```

Result Sets

- **A query yields a ResultSet:**

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books")
```

- **Use this loop to iterate over the rows:**

```
while (rs.next())
{
    look at a row of the result set
}
```

- **To get at the columns of a row, use one of the get methods:**

```
String isbn = rs.getString(1); // The first (!) column
double price = rs.getDouble("Price");
```

BipinRupadiya.com

Prepared Statements

- **When a query has variable parts, you don't want to formulate it through string concatenation:**

```
String query = "SELECT * FROM Books WHERE Books.Title = " + title;
   // Don't—or you may become the victim of SQL injection
```

- **Instead, use a prepared statement:**

```
String query = "SELECT * FROM Books WHERE Books.Title = ?";
PreparedStatement stat = conn.prepareStatement(query);
stat.setString(1, title);
ResultSet rs = stat.executeQuery();
```

- **A PreparedStatement becomes invalid after closing the Connection that created it.**
  - ❑ But the database will cache the query plan.

BipinRupadiya.com

Retrieving Autogenerated Values

- **Most databases support some mechanism for automatically generating values.**

  MySQL: Id INTEGER AUTO_INCREMENT

  Derby : Id INTEGER GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1)

- **You can retrieve the autogenerated values that were generated by an INSERT statement:**

```
stat.executeUpdate(insertStatement, tatement.RETURN_GENERATED_KEYS);
ResultSet rs = stat.getGeneratedKeys();
if (rs.next()) {
    int key = rs.getInt(1);
    Do something with key
}
```

Contact:

Bipin S. Rupadiya

Assistant Professor, JVIMS

Mo. : +91-9228582425

Email: info@bipinrupadiya.com

**Blog : ** www.BipinRupadiya.com