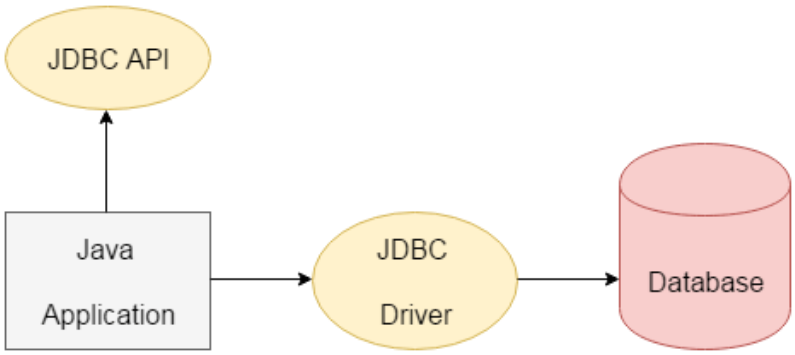
# ADVANCED JAVA - UNIT I: JDBC

## 1. JDBC: Introduction

- JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database.
- **Java Database Connectivity (JDBC)** is an **Application Programming Interface(API)** used to connect Java application with Database.
- JDBC is used to interact with various type of Database such as Oracle, MS Access, My SQL and SQL Server.
- JDBC can also be defined as the platform-independent interface between a relational database and Java programming.
- It allows java program to execute SQL statement and retrieve result from database.




JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

### 1.1 API

API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

A list of popular *interfaces* of JDBC API are given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

## 1.2 JDBC Components

JDBC includes four components:

### 1. The JDBC API

The JDBC™ API provides programmatic access to relational data from the Java™ programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.

The JDBC API is part of the Java platform, which includes the *Java™ Standard Edition* (Java™ SE ) and the *Java™ Enterprise Edition* (Java™ EE). The JDBC 4.0 API is divided into two packages: java.sql and javax.sql. Both packages are included in the Java SE and Java EE platforms.

### 2. JDBC Driver Manager

The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver.

DriverManager has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

The Standard Extension packages javax.naming and javax.sql let you use a Data Source object registered with a *Java Naming and Directory Interface*™ (JNDI) naming service to establish a connection with a data source. You can use either connecting mechanism, but using a DataSource object is recommended whenever possible.

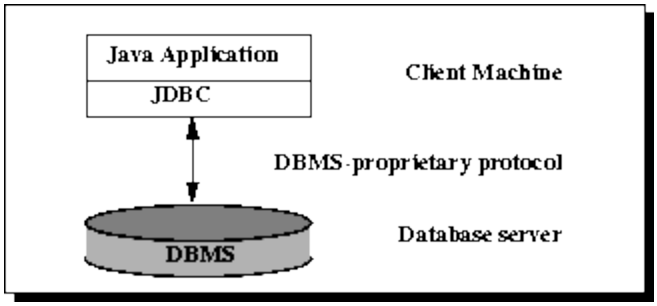
**3. JDBC Test Suite**

The JDBC driver test suite helps you to determine that JDBC drivers will run your program.

4. **JDBC-ODBC Bridge**

The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

## 1.3 JDBC Architecture

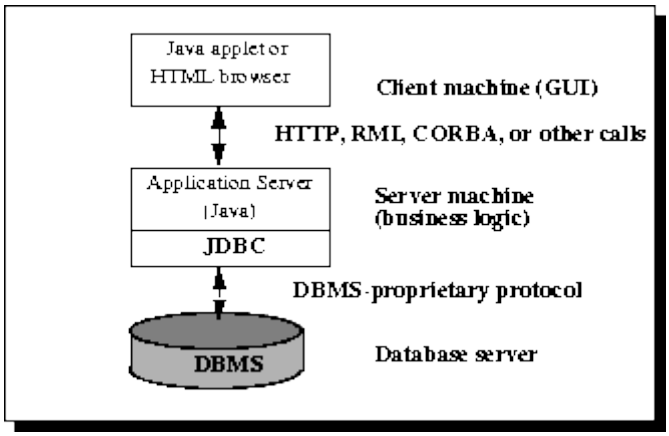



In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed.

A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-

tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.




Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

### 1.4 JDBC Vs ODBC

| ODBC | JDBC |
|---|---|
| ODBC Stands for Open Database Connectivity. | JDBC Stands for java database connectivity. |
| Introduced by Microsoft in 1992. | Introduced by SUN Micro Systems in 1997. |

| | |
|---|---|
| We can use ODBC for any language like C,C++,Java etc. | We can use JDBC only for Java languages. |
| We can choose ODBC only windows platform. | We can Use JDBC in any platform. |
| Mostly ODBC Driver developed in native languages like C,C++. | JDBC Stands for java database connectivity. |
| For Java applications it is not recommended to use ODBC because performance will be down due to internal conversion and applications will become platform Dependent. | For Java application it is highly recommended to use JDBC because there we no performance & platform dependent problem. |
| ODBC is procedural. | JDBC is object oriented |

### 1.5 JDBC Driver Model

DBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

#### 1. JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

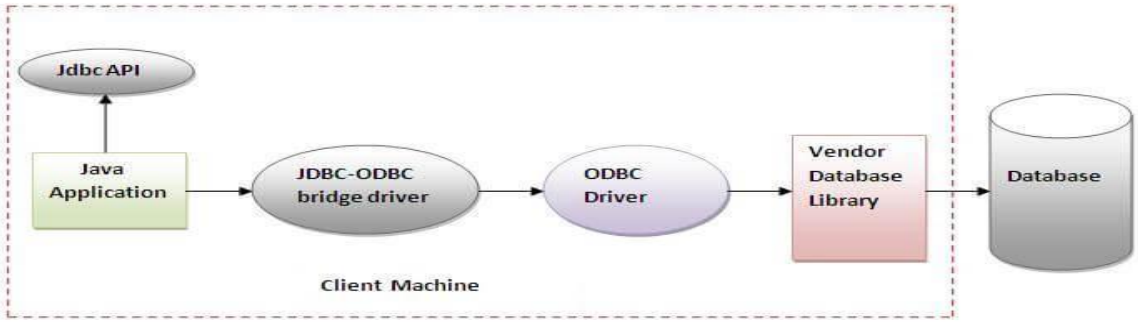



Figure- JDBC-ODBC Bridge Driver

**Advantages:**

- Easy to use.
- Can be easily connected to any database.

**Disadvantages:**

- Performance degraded because JDBC method call is converted into the ODBC function calls. The ODBC driver needs to be installed on the client machine.

**2) Native-API driver**

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

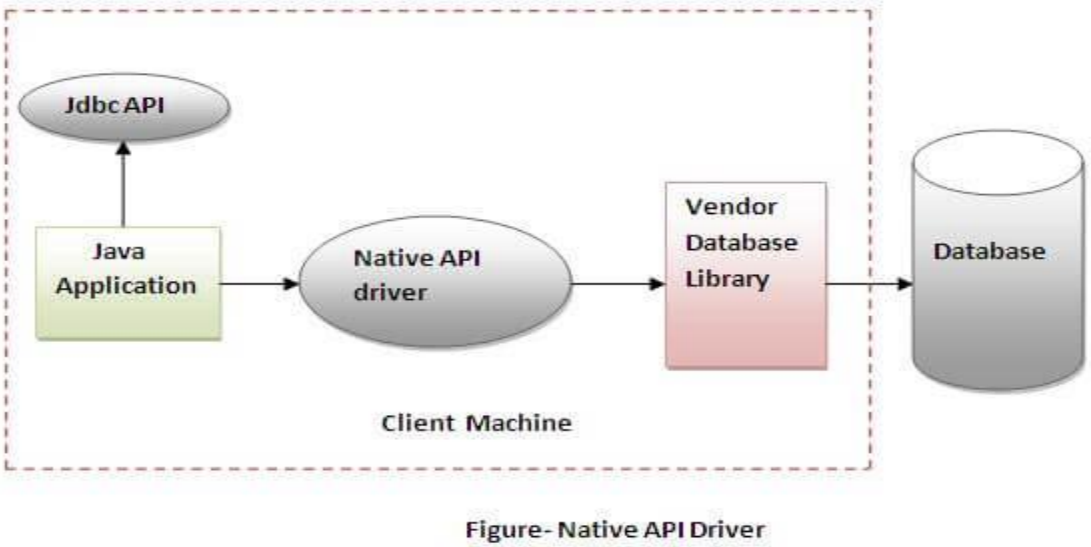



Figure- Native API Driver

Native-API driver

**Advantage:**

- Performance upgraded than JDBC-ODBC bridge driver.

**Disadvantage:**

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

**3) Network Protocol driver**

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

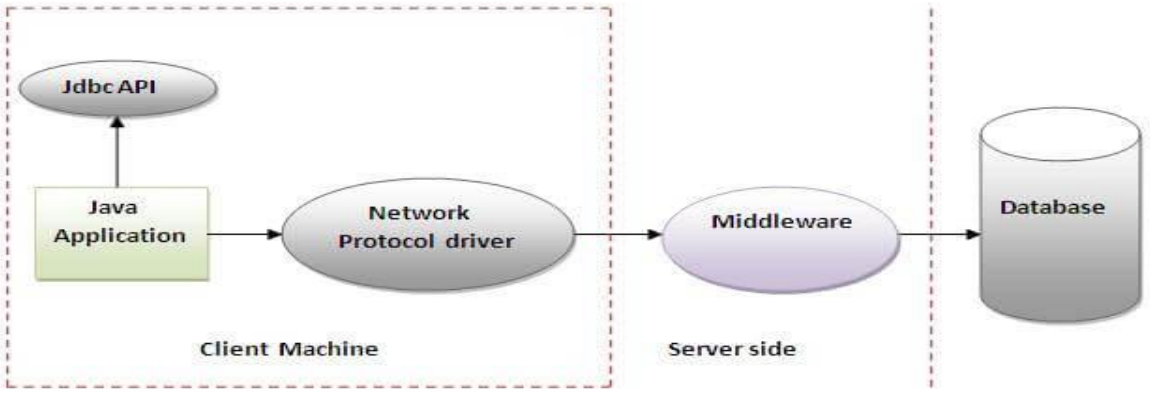



Figure- Network Protocol Driver

**Advantage:**

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

**Disadvantages:**

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

**4) Thin driver**

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

**Advantage:**

- Better performance than all other drivers.
- No software is required at client side or server side.

**Disadvantage:**

- Drivers depend on the Database.JAVA.SQL Packages

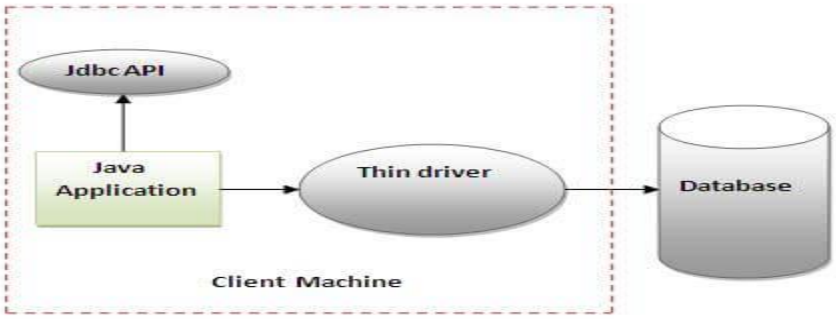



Figure- Thin Driver

### 1.6 JAVA.SQL Package

This package include classes and interface to perform almost all JDBC operation such as creating and executing SQL Queries.

**important classes and interface of `java.sql` package**

| classes/interface | Description |
|---|---|
| `java.sql.BLOB` | Provide support for BLOB(Binary Large Object) SQL type. |
| `java.sql.Connection` | creates a connection with specific database |
| `java.sql.CallableStatement` | Execute stored procedures |
| `java.sql.CLOB` | Provide support for CLOB(Character Large Object) SQL type. |
| `java.sql.Date` | Provide support for Date SQL type. |
| `java.sql.Driver` | create an instance of a driver with the DriverManager. |
| `java.sql.DriverManager` | This class manages database drivers. |
| `java.sql.PreparedStatement` | Used to create and execute parameterized query. |
| `java.sql.ResultSet` | It is an interface that provide methods to access the result row-by-row. |
| `java.sql.Savepoint` | Specify savepoint in transaction. |
| `java.sql.SQLException` | Encapsulate all JDBC related exception. |
| `java.sql.Statement` | This interface is used to execute SQL statements. |

**1. java.sql.BLOB**

- Provide support for BLOB(Binary Large Object) SQL type.
- A **Binary Large OBject** (**BLOB**) is a collection of binary data stored as a single entity in a database management system. Blobs are typically images, audio or other multimediaobjects, though sometimes binary executable code is stored as a blob. Database support for blobs is not universal.

**Syntax:**

public interface Blob

- An SQL BLOB is a built-in type that stores a Binary Large Object as a column value in a row of a database table.
- By default drivers implement Blob using an SQL locator(BLOB), which means that a Blob object contains a logical pointer to the SQL BLOB data rather than the data itself.

- A Blob object is valid for the duration of the transaction in which is was created.

2. java.sql.CallableStatement:

- Execute stored procedures
- The interface used to execute SQL stored procedures. The JDBC API provides a stored procedure SQL escape syntax that allows stored procedures to be called in a standard way for all RDBMSs.

**Syntax:**

public interface CallableStatement extends PreparedStatement

- A CallableStatement can return one ResultSet object or multiple ResultSet objects. Multiple ResultSet objects are handled using operations inherited from Statement.

3. java.sql.Connection

- Creates a connection with specific database
- A connection (session) with a specific database. SQL statements are executed and results are returned within the context of a connection.

- A Connection object's database is able to provide information describing its tables, its supported SQL grammar, its stored procedures, the capabilities of this connection, and so on. This information is obtained with the getMetaData method.

4. java.sql.CLOB

- Provide support for CLOB(Character Large Object) SQL type.
- An SQL CLOB is a built-in type that stores a Character Large Object as a column value in a row of a database table. By default drivers implement a Clob object using an SQL locator(CLOB), which means that a Clob object contains a logical pointer to the SQL CLOB data rather than the data itself. A Clob object is valid for the duration of the transaction in which it was created.
- The Clob interface provides methods for getting the length of an SQL CLOB (Character Large Object) value, for materializing a CLOB value on the client, and for searching for a substring or CLOB object within a CLOB value. Methods in the interfaces ResultSet, CallableStatement, and PreparedStatement, such as getClob and setClob allow a programmer to access an SQL CLOB value. In addition, this interface has methods for updating a CLOB value.

**Example:**

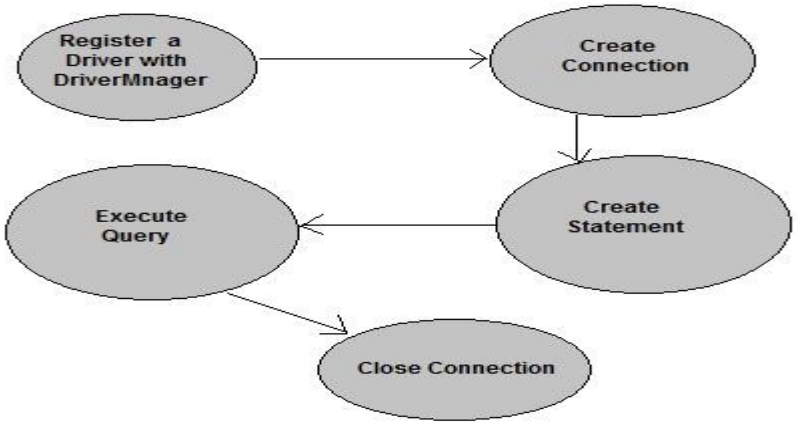
CREATE TABLE countries ( name VARCHAR2(90), notes CLO );

Insert some data

INSERT INTO countries VALUES ('Greece', 'Greece is a country in south-east Europe. Athens is the capital...');

### 1.7 JDBC Connection

**Steps to connect a Java Application to Database**

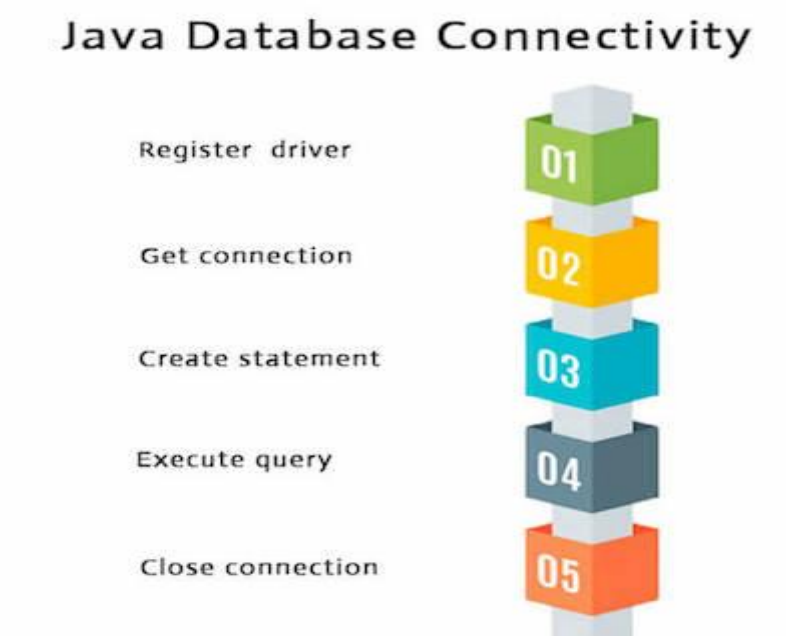



Block diagram of Database Connectivity

**Java Database Connectivity with 5 Steps**

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection



**1) Register the driver class**

The forName() method of class is used to register the driver class. This method is used to dynamically load the driver class.

**Syntax of forName() method**

public static void forName(String className)throws ClassNotFoundException

**Example to register the OracleDriver class**

Here, Java program is loading oracle driver to esteblish database connection.

```
1. Class.forName("oracle.jdbc.driver.OracleDriver");
```

**2) Create the connection object**

The **getConnection()** method of DriverManager class is used to establish connection with the database

**Syntax of getConnection() method**

public static Connection getConnection(String url)throws SQLException

public static Connection getConnection(String url,String name,String password) throws SQLException

**Example to establish connection with the Oracle database**

```
1. Connection con=DriverManager.getConnection(
2. "jdbc:oracle:thin:@localhost:1521:xe","system","password");
```

**3) Create the Statement object**

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

**Syntax of createStatement() method**

public Statement createStatement()throws SQLException

**Example to create the statement object**

```
Statement stmt=con.createStatement();
```

**4) Execute the query**

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

**Syntax of executeQuery() method**

public ResultSet executeQuery(String sql)throws SQLException

**Example to execute query**

```
1. ResultSet rs=stmt.executeQuery("select * from emp");
2.
3. while(rs.next()){
4. System.out.println(rs.getInt(1)+" "+rs.getString(2));
5. }
```

**5) Close the connection object**

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection

**Syntax of close() method**

```
1. public void close()throws SQLException
```

**Example to close connection**

```
1. con.close();
```

(Take Lab exercise as Example)

### 1.8 Prepared Statement and Callable Statement

#### 1.8.1 Statement

- ✓ The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet.
- ✓ The object used for executing a static SQL statement and returning the results it produces.
- ✓ By default, only one `ResultSet` object per `Statement` object can be open at the same time. Therefore, if the reading of one `ResultSet` object is interleaved with the reading of another, each must have been generated by different `Statement`

objects. All execution methods in the `Statement` interface implicitly close a statment's current `ResultSet` object if an open one exists.

### 1.8.2 Resultset

- ✓ A table of data representing a database result set, which is usually generated by executing a statement that queries the database.
- ✓ A `ResultSet` object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The `next` method moves the cursor to the next row, and because it returns `false` when there are no more rows in the `ResultSet` object, it can be used in a `while` loop to iterate through the result set.
- ✓ A default `ResultSet` object is not updatable and has a cursor that moves forward only. Thus, you can iterate through it only once and only from the first row to the last row. It is possible to produce `ResultSet` objects that are scrollable and/or updatable. The following code fragment, in which `con` is a valid `Connection` object, illustrates how to make a result set that is scrollable and insensitive to updates by others, and that is updatable. See `ResultSet` fields for other options.

## Commonly used methods of Statement interface:

The important methods of Statement interface are as follows:

**1) public ResultSet executeQuery(String sql)**: is used to execute SELECT query. It returns the object of ResultSet.

**2) public int executeUpdate(String sql):** is used to execute specified query, it may be create, drop, insert, update, delete etc.

**3) public boolean execute(String sql):** is used to execute queries that may return multiple results.

**4) public int[] executeBatch():** is used to execute batch of commands. The important methods of Statement interface are as follows:

### 1.8.3 CallableStatement

- ✓ CallableStatement interface is used to call the **stored procedures and functions**.

- ✓ We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.

The differences between stored procedures and functions are given below:

| Stored Procedure | Function |
|---|---|
| is used to perform business logic. | is used to perform calculation. |
| must not have the return type. | must have the return type. |
| may return 0 or more values. | may return only one values. |
| We can call functions from the procedure. | Procedure cannot be called from function. |
| Procedure supports input and output parameters. | Function supports only input parameter. |
| Exception handling using try/catch block can be used in stored procedures. | Exception handling using try/catch can't be used in user defined functions. |

### 1.8.4 Prepared Statement

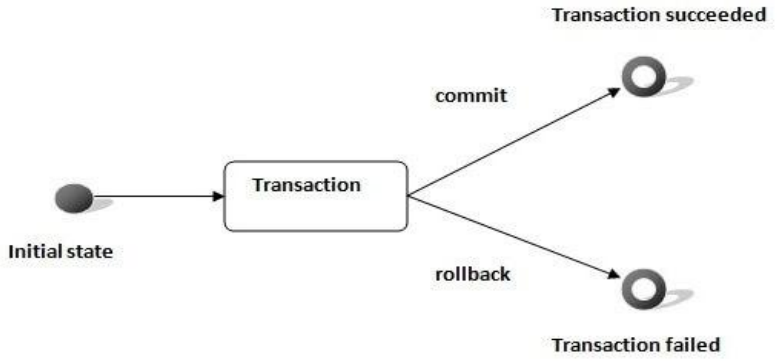
- ✓ The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.
- ✓ An object that represents a precompiled SQL statement.
- ✓ A SQL statement is precompiled and stored in a `PreparedStatement` object. This object can then be used to efficiently execute this statement multiple times.

**Example:**

```
PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES
    SET SALARY = ? WHERE ID = ?");
pstmt.setBigDecimal(1, 153833.00)
pstmt.setInt(2, 110592)
```

### 1.9 Transactions

- ✓ Transaction represents **a single unit of work**.
- ✓ The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.
  - ➢ **Atomicity** means either all successful or none.
  - ➢ **Consistency** ensures bringing the database from one consistent state to another consistent state.
  - ➢ **Isolation** ensures that transaction is isolated from other transaction.
  - ➢ **Durability** means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.




Transaction Management

### 1.10 Cursor & Batch Update

**Cursors in SQL**

- A cursor is a temporary work area created in system memory when an SQL statement is executed.
- A cursor is a set of rows together with a pointer that identifies a current row. It is a database object to retrieve data from a **result set** one row at a time.
- In other words, a cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the active set.

**Types of Cursors**

There are the following two types of Cursors:

1. Implicit Cursor
2. Explicit Cursor

**Implicit Cursor**

- ✓ These types of cursors are generated and used by the system during the manipulation of a DML query (INSERT, UPDATE and DELETE). An implicit cursor is also generated by the system when a single row is selected by a SELECT command.

**Explicit Cursor**

- ✓ This type of cursor is generated by the user using a SELECT command. An explicit cursor contains more than one row, but only one row can be processed at a time. An explicit cursor moves one by one over the records. An explicit cursor uses a pointer that holds the record of a row. After fetching a row, the cursor pointer moves to the next row.

**Main components of Cursors**

Each cursor contains the followings 5 parts:

1. **Declare Cursor:** In this part we declare variables and return a set of values.
2. **Open:** This is the entering part of the cursor.
3. **Fetch:** Used to retrieve the data row by row from a cursor.
4. **Close:** This is an exit part of the cursor and used to close a cursor.
5. **Deallocate:** In this part we delete the cursor definition and release all the system resources associated with the cursor.

**1.11 Batch Processing:**

- ✓ Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast.
- ✓ The java.sql.Statement and java.sql.PreparedStatement interfaces provide methods for batch processing.

**Creating Batch Processing:**

. It follows following steps:

- Load the driver class
- Create Connection
- Create Statement
- Add query in the batch
- Execute Batch
- Close Connection

**(Take Lab exercise as a example)**

**1.12 RMI Basic – Client/ Server Application**

Remote method invocation(RMI) allow a java object to invoke method on an object running on another machine. RMI provide remote communication between java program. RMI is used for building distributed application.

**Concept of RMI application**

A RMI application can be divided into two part, **Client** program and **Server** program. A **Server** program creates some remote object, make their references available for the client to invoke method on it.

A **Client** program make request for remote objects on server and invoke method on them. **Stub** and **Skeleton** are two important object used for communication with remote object.

## Stub

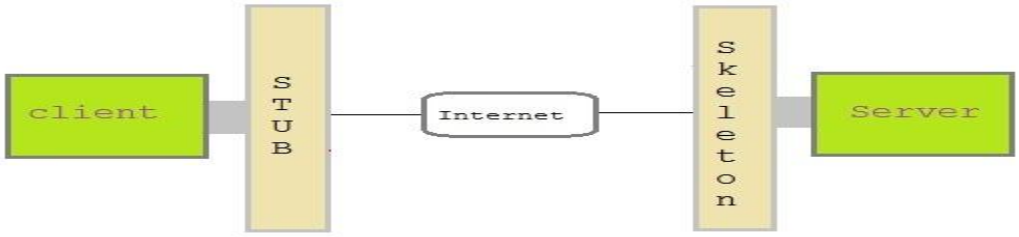
The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

## Skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.



**Creating a Simple RMI application involves following steps**

- Define a remote interface.
- Implementing remote interface.
- create and start remote application
- create and start client application

**Define a remote interface**

A remote interface specifies the methods that can be invoked remotely by a client. Clients program communicate to remote interfaces, not to classes implementing it. To be a remote interface, a interface must extend the **Remote** interface of **java.rmi** package.

```
import java.rmi.*;
public interface AddServerInterface extends Remote
{
public int sum(int a,int b);
}
```

**Implementation of remote interface**

For implementation of remote interface, a class must either extend **UnicastRemoteObject** or use exportObject() method of **UnicastRemoteObject** class.

```
import java.rmi.*;
import java.rmi.server.*;
public class Adder extends UnicastRemoteObject implements AddServerInterface
{
Adder()throws RemoteException{
super();
}
public int sum(int a,int b)
{
return a+b;
}
}
```

**Create AddServer and host rmi service**

You need to create a server application and host rmi service **Adder** in it. This is done using rebind() method of **java.rmi.Naming** class. rebind() method take two arguments, first represent the name of the object reference and second argument is reference to instance of Adder

**Create client application**

Client application contains a java program that invokes the lookup() method of the **Naming** class. This method accepts one argument, the **rmi** URL and returns a reference to an object of type **AddServerInterface**. All remote method invocation is done on this object.

```
import java.rmi.*;
public class Client{
public static void main(String args[]){
try{
AddServerInterface
st=(AddServerInterface)Naming.lookup("rmi://"+args[0]+"/AddService");
System.out.println(st.sum(25,8));
}catch(Exception e){System.out.println(e);}
}
}
```

**Steps to run this RMI application**

Save all the above java file into a directory and name it as "rmi"

- compile all the java files
  - javac *.java
- Start RMI registry
  - `start rmiregistry`
- Run Server file
  - `java AddServer`
- Run Client file in another command prompt abd pass local host port number at run time
  - `java Client 127.0.0.1`

*******