

**SRINIVAS UNIVERSITY**

**MANGALORE**

**INSTITUTE OF COMPUTER SCIENCE AND INFORMATION SCIENCE**

**Course Name: MCA I Year: Semester Name: II Semester**

**Subject Name: Advanced Java**

## **ADVANCED JAVA**

### **UNIT – 1**

JDBC: Introduction to JDBC, JDBC Driver types, JDBC database connections, JDBC Statements, PreparedStatement, CallableStatement, ResultSet, JDBC data types, transactions, Batch Processing, Stored Procedure

**Advanced Java: Introduction to advanced Java,** Java is a general-purpose high-level programming language that helps to build a variety of applications. Java is popular as it provides platform as it provides various features such as independency, security, multithread support.

There are two types of Java as Core Java and Advanced Java. Core Java covers the fundamental concepts in the Java programming language. On the other hand, Advanced Java is the next level after Core Java.

The Core Java is used to build general applications while the Advanced Java is used to build enterprise level applications. Advance Java i.e. JEE (Java Enterprise Edition) gives you the library to understand the Client-Server architecture for Web Application Development which Core Java doesn't support.

J2EE is platform Independent, Java Centric environment for developing, building & deploying Web-based applications online. It also consists of a set of services, APIs, and protocols, which provides the functionality that is necessary for developing multi-tiered, web-based applications.

## Example: Comparison between the core Java vs advanced Java

Core java	Advanced Java
Includes Java Standard Edition (J2SE)	Includes Java Enterprise Edition (J2EE)
Category of java that covers the fundamental concepts of Java programming language to develop general applications	Category of java that covers the advanced concepts to build enterprise applications using Java programming language
OOP, data types, operators, exception handling, threading, swing and collection are some topics	<u>Database connectivity, Web services, Servlets, JSP, EJB and some topics</u>
Uses single tier architecture . single-tier applications are desktop applications like MS Office, PC Games, image editing software like Gimp, Photoshop	Uses two tier architecture client and server architecture. presentation layer runs on a client and data is stored on a server. PC, Mobile, Tablet,
Helps to build general applications	Helps to build enterprise level application .web applications

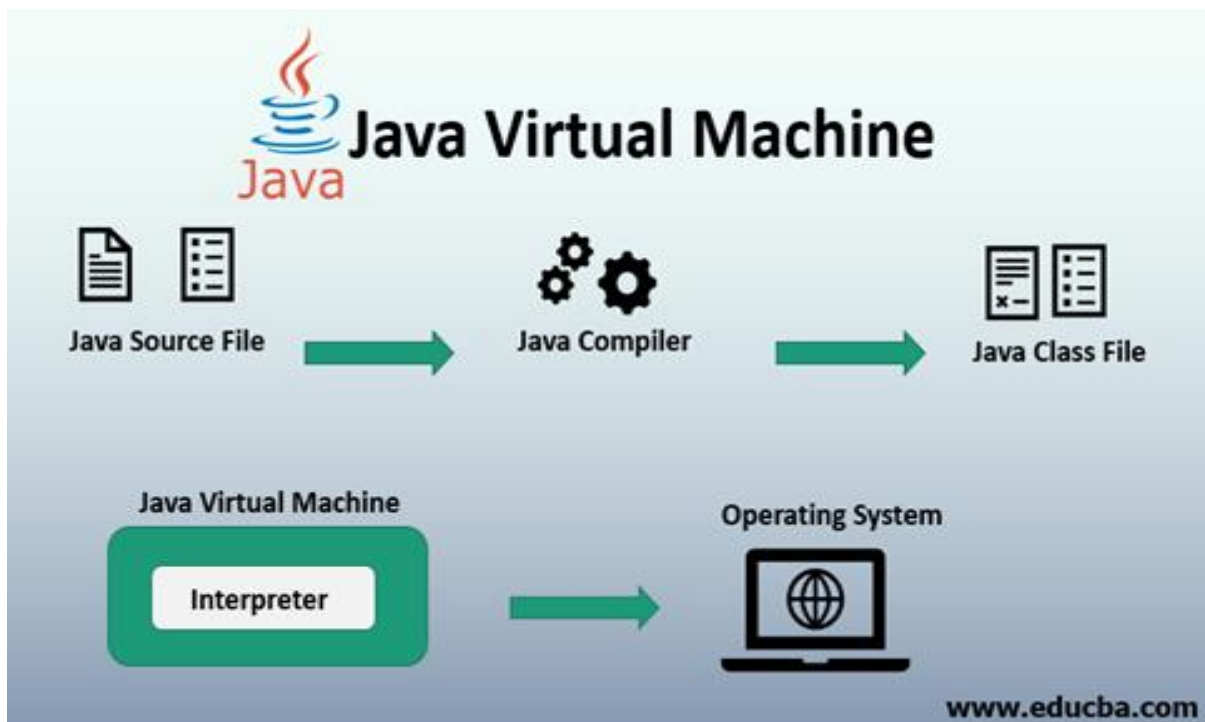
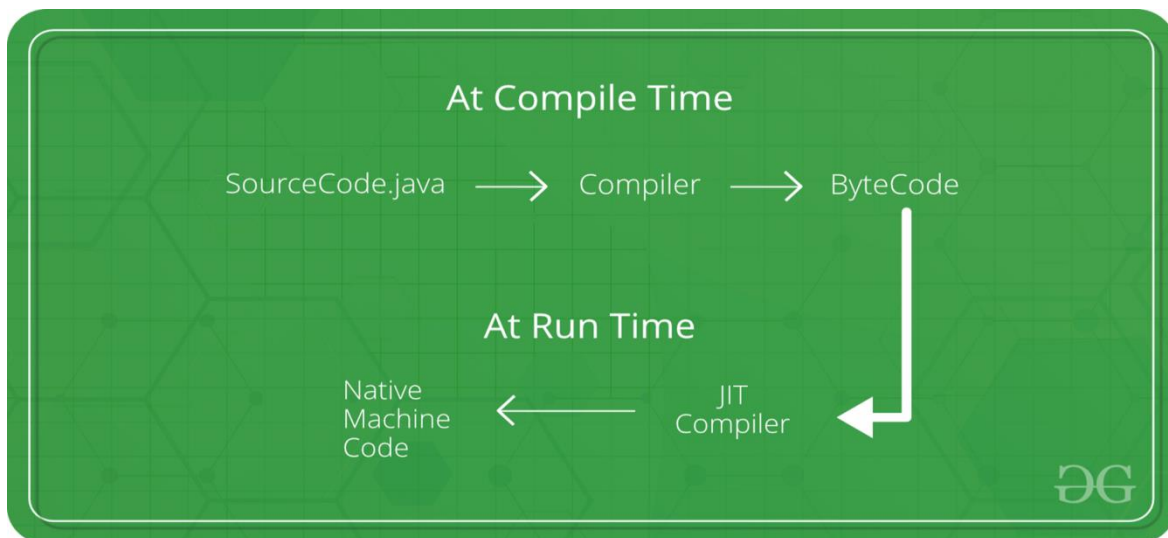
Advanced Java is the level ahead of Core Java, and covers more advanced concepts such as web technologies, and database accessing. Java Enterprise Edition (J2EE) is categorized as Advanced Java.

Advanced Java covers a number of topics. JDBC stands for Java Database Connectivity. It is a standard Java API to build independent connectivity between the Java language based application and databases such as MySQL My structured query language, MSSQL Microsoft Sql server, and, Oracle. Additionally, Servlets and JSP allow developing dynamic web applications. EJB provide distributed and highly transactional features to build enterprise applications. Furthermore, Java web services help to build SOAP simple object access protocol and REST Representational state transfer Architecture style expose the url/users full web services. They provide a common platform for the applications to communicate with each other.

## JAVA Virtual Machine

The Just-In-Time (JIT) compiler is a component of the runtime environment that improves the performance of Java™ applications by compiling bytecodes to native machine code at run time.

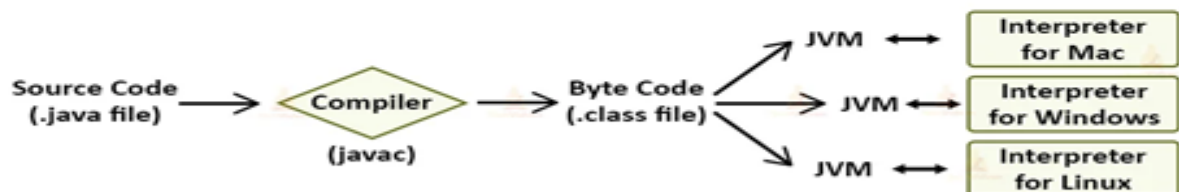
JVM compiles complete byte code to machine code. JIT compiles only the reusable byte code to machine code. JVM provides platform independence. JIT improves the performance of JVM.



# Java Virtual Machine

```
public class MyFirstJavaProgram {  
    public static void main(String [] args) {  
        System.out.println("Hello World");  
    }  
}
```

C:\> javac MyFirstJavaProgram.java -> Compile  
C:\> java MyFirstJavaProgram -> Run the program  
Hello World



Java programs are first compiled into Java Byte Code(Binary form) and then a special Java interpreter interprets them for a specific platform. Java ByteCode is the machine language for Java Virtual machine(JVM). The JVM converts the compiled binary byte code into a specific machine language.

## Compile time vs Run time

Runtime and compile time, these are two programming terms that are more frequently used in java programming language. The programmers specially beginners find it little difficult to understand what exactly they are. So let's understand what these terms means in java with example.

In java running a program happens in two steps, compilation and then execution. The image below shows where does compile time and runtime takes place in execution of a program.

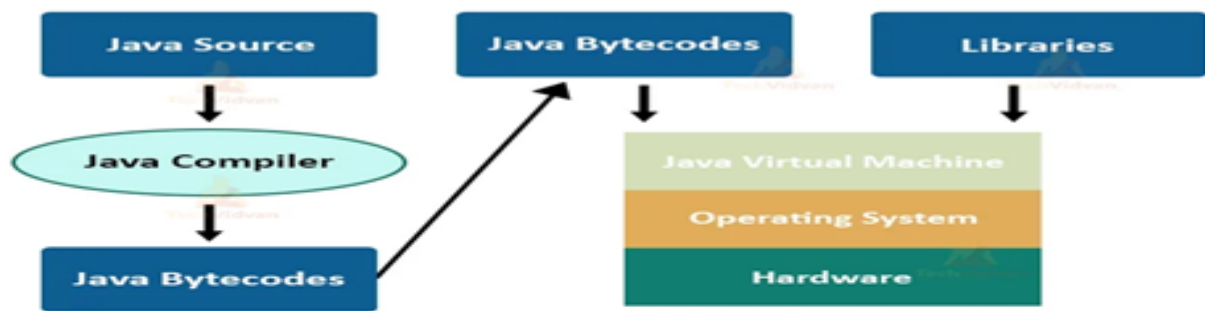


As soon as the programmer starts executing the program using java command, runtime gets started and it ends when execution of program ended either successfully or unsuccessfully. In other way the process of running a program is known as runtime.

### Difference between runtime and compile time

Compile time is a process in which java compiler compiles the java program and generates a .class file. In other way, in compile time java source code(.java file) is converted into .class file using java compiler. While in runtime, the java virtual machine loads the .class file in memory and executes that class to generate the output of program.

## Working of JVM



JVM(Java Virtual Machine) behaves as a run-time engine to run Java applications. JVM calls the main method present in Java code.

Java Virtual machine(JVM) is a part of the JRE(Java Runtime Environment).

Java applications are WORA (Write Once Run Anywhere). This means that we need to write the Java programs just once, which we can run on different platforms without making changes in the Java program.

When we compile a .java file, the compiler generates the .class files (contains byte-code) with the same names as that of the class present in a .java file.

When we run a .class file, it goes through various steps.

### Advanced Java applications

There are a wide range of applications for advanced Java. Typically, programmers use it for web and network-focused applications and databases. Some of its applications include:

**Mobile:** Java is popular with mobile app developers because of its compatibility range.

**Graphical user interfaces (GUIs):** When developing GUIs within corporate networks, programmers often use advanced Java .

**Web:** Advanced Java is a popular choice for web applications, as it's easy to use and has a high level of security.

**Enterprise:** Developers of enterprise applications, such as banking applications, often use advanced Java because of its advantageous runtime environment and compatibility with web services.

**Scientific:** Advanced Java is a popular choice for developers for coding mathematical and scientific calculations.

**Gaming:** Game developers often use advanced Java for designing 3D games.

**Big data:** Databases commonly use advanced Java to help organize large volumes of information.

**Distributed applications:** Developers frequently use Java for distributed applications because of its persistent and dynamic nature.

**Cloud-based applications:** Java is a popular choice for cloud-based applications, as it's compatible with software as a service (SaaS) and similar applications for platforms (PaaS) and infrastructure (IaaS).

**Example:**

Java Platform, Micro Edition (Java ME) provides a robust, flexible environment for applications running on embedded and mobile devices in the Internet of Things: micro-controllers, sensors, gateways, mobile phones, personal digital assistants (PDAs), TV set-top boxes, printers and more...

## **Introduction to JDBC**

JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- a. Making a connection to a database.
- b. Creating SQL or MySQL statements.
- c. Executing SQL or MySQL queries in the database.
- d. Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executable, such as –

## 1.Java Applications

## 2.Java Applets

## 3.Java Servlets

## 4.Java ServerPages (JSPs)

## 5.Enterprise JavaBeans (EJBs).

All of these different executable are able to use a JDBC driver to access a database, and take advantage of the stored data. JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

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

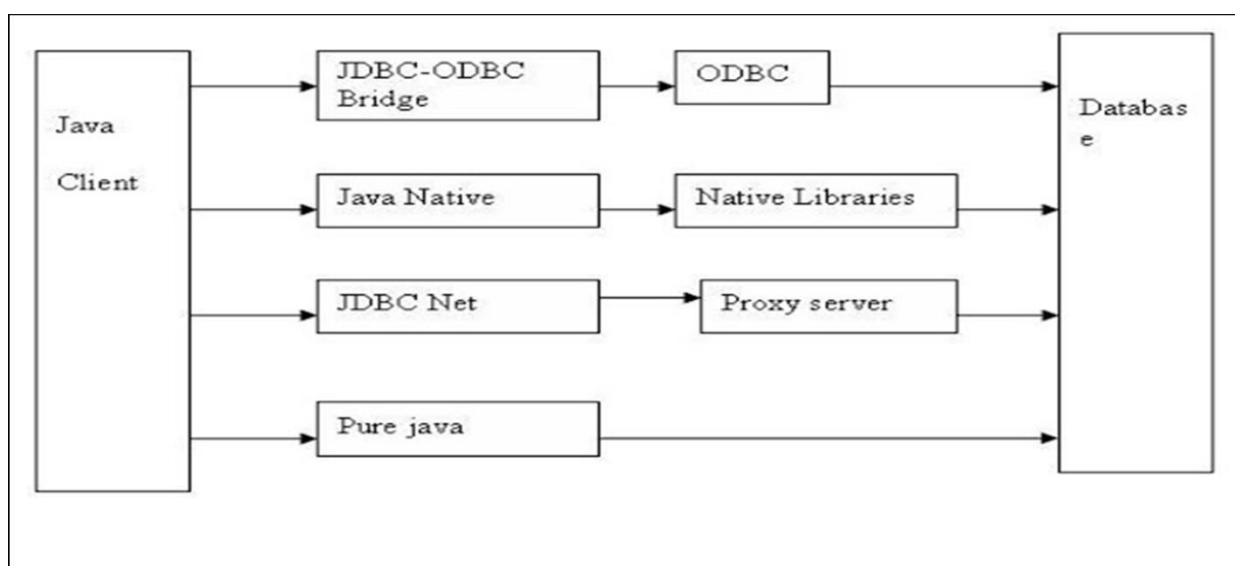
### **JDBC-ODBC bridge driver**

### **Native-API driver (partially java driver)**

### **Network Protocol driver (fully java driver)**

### **Thin driver (fully java driver)**

**Note:** The Type 1 JDBC driver is simply a JDBC-ODBC bridge. The Type 2 JDBC driver is written in a language other than Java, often C++ or C. The Type 3 JDBC driver talks to a middleware server first, not the database directly. The Type 4 JDBC driver is a pure, direct Java-to-the-database implementation.





## **Define JDBC Concept with example.**

JDBC or Java Database Connectivity is a Java API to connect and execute the query with the database. It is a specification from Sun microsystems that provides a standard abstraction(API or Protocol) for java applications to communicate with various databases. It provides the language with java database connectivity standards. It is used to write programs required to access databases. JDBC, along with the database driver, can access databases and spreadsheets. The enterprise data stored in a relational database(RDB) can be accessed with the help of JDBC APIs.

### **Definition of JDBC(Java Database Connectivity)**

JDBC is an API(Application programming interface) used in java programming to interact with databases. The classes and interfaces of JDBC allow the application to send requests made by users to the specified database.

## **Purpose of JDBC**

Enterprise applications created using the JAVA EE technology need to interact with databases to store application-specific information. So, interacting with a database requires efficient database connectivity, which can be achieved by using the ODBC(Open database connectivity) driver. This driver is used with JDBC to interact or communicate with various kinds of databases such as Oracle, MS Access, Mysql, and SQL server database.

## **Components of JDBC**

There are generally four main components of JDBC through which it can interact with a database. They are as mentioned below:

**1. JDBC API:** It provides various methods and interfaces for easy communication with the database. It provides two packages as follows, which contain the java SE and Java EE platforms to exhibit WORA(write once run anywhere) capabilities.

```
java.sql.*;
```

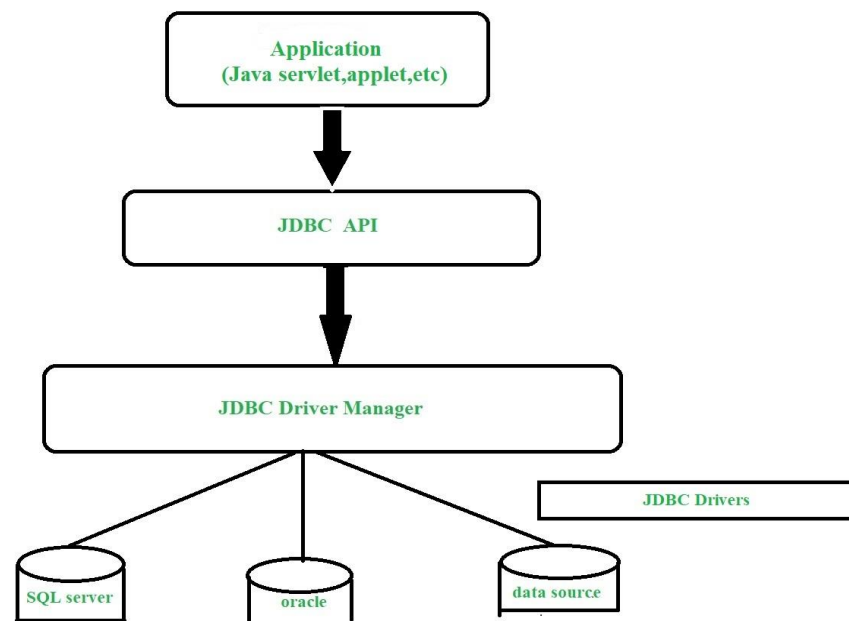
It also provides a standard to connect a database to a client application.

**2. JDBC Driver manager:** It loads a database-specific driver in an application to establish a connection with a database. It is used to make a database-specific call to the database to process the user request.



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. It makes use of the **sun.jdbc.odbc** package which includes a native library to access ODBC characteristics.

### Architecture of JDBC



### **Description:**

1. **Application:** It is a java applet or a servlet that communicates with a data source.
2. **The JDBC API:** The JDBC API allows Java programs to execute SQL statements and retrieve results. Some of the important classes and interfaces defined in JDBC API are as follows:
3. **DriverManager:** It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.
4. **JDBC drivers:** To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

## JDBC Drivers

[JDBC drivers](#) are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. 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)

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

The JDBC architecture consists of [two-tier and three-tier processing models](#) to access a database. They are as described below:

1. **Two-tier model:** A java application communicates directly to the data source. The JDBC driver enables the communication between the application and the data source. When a user sends a query to the data source, the answers for those queries are sent back to the user in the form of results.  
The data source can be located on a different machine on a network to which a user is connected. This is known as a **client/server configuration**, where the user's machine acts as a client, and the machine has the data source running acts as the server.
2. **Three-tier model:** In this, the user's queries are sent to middle-tier services, from which the commands are again sent to the data source. The results are sent back to the middle tier, and from there to the user. This type of model is found very useful by management information system directors.

## Interfaces of JDBC API

A list of popular *interfaces* of JDBC API is given below of example

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

JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server. Essentially, a JDBC driver makes it possible to do three things:

1. Establish a connection with a data source.
2. Send queries and update statements to the data source.
3. Process the results.

For example, the use of JDBC drivers enables you to open a database connection to interact with it by sending SQL or database commands.

## Working of JDBC

Java application that needs to communicate with the database has to be programmed using JDBC API. JDBC Driver supporting data sources such as Oracle and SQL server has to be added in java application for JDBC support which can be done dynamically at run time. This JDBC driver intelligently communicates the respective data source.

Example: Creating a simple JDBC application : Java Program

```
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();
    }
} // class
```

The above example demonstrates the basic steps to access a database using JDBC. The application uses the JDBC-ODBC bridge driver to connect to the database. You must import **java.sql** package to provide basic SQL functionality and use the classes of the package.

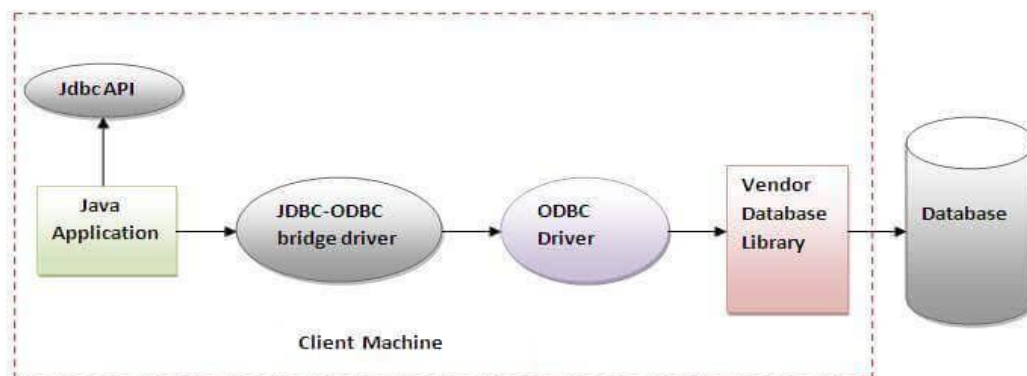
#### **JDBC Driver Types with example.**

JDBC 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**

In Java 8, the JDBC-ODBC Bridge has been removed.

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

### 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.

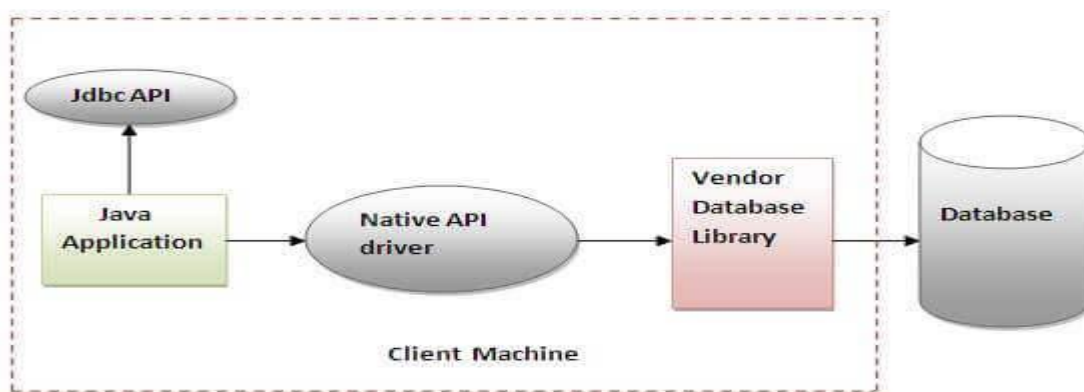


Figure- Native API Driver

### 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.

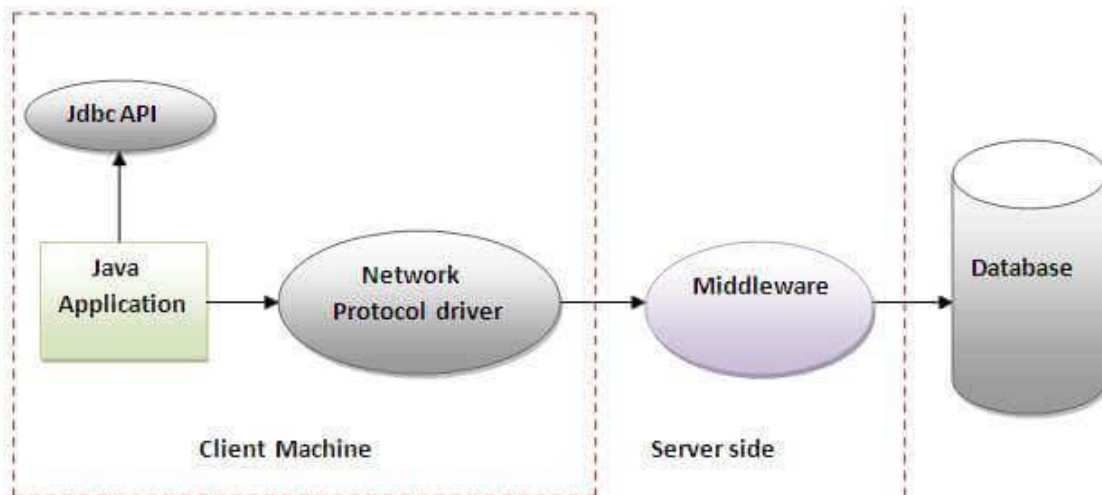


Figure- Network Protocol Driver

#### 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.

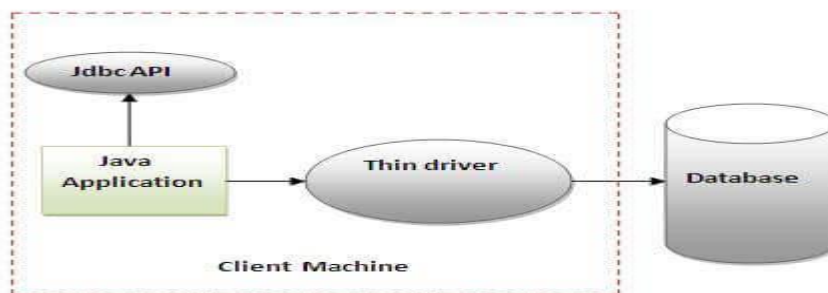


Figure- Thin Driver

### Advantage:

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

### Disadvantage:

- Drivers depend on the Database.

### JDBC Driver Performance

Property	Type-1	Type-2	Type-3	Type-4
Conversion	From JDBC calls to ODBC calls	From JDBC calls to native library calls	From JDBC calls to middle-wear specific calls	From JDBC calls to Data Base specific calls
Implemented-in	Only java	Java + Native language	Only java	Only java
Architecture	Follow 2-tier architecture	Follow 2-tier architecture	Follow 3-tier architecture	Follow 2-tier architecture
Platform-independent	NO	NO	YES	YES
Data Base independent	YES	NO	YES	NO
Thin or Thick	Thick	Thick	Thick	Thin



## JDBC Database Connection:

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

5 Steps to connect to the database in java

1. Register the driver class
2. Create the connection object
3. Create the Statement object
4. Execute the query
5. Close the connection object

### 1) Register the driver class

The `forName()` method of `Class` 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`

Note: Since JDBC 4.0, explicitly registering the driver is optional. We just need to put vendor's Jar in the classpath, and then JDBC driver manager can detect and load the driver automatically.

### Example to register the OracleDriver class

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

```
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

1) `public static Connection getConnection(String url)`**throws** `SQLException`

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

## Example to establish connection with the Oracle database

```
Connection con=DriverManager.getConnection("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

```
ResultSet rs=stmt.executeQuery("select * from emp");
```

```
while(rs.next()){
```

```
System.out.println(rs.getInt(1)+" "+rs.getString(2));
```

```
}
```

### **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

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

#### **Example to close connection**

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

Note: Since Java 7, JDBC has ability to use try-with-resources statement to automatically close resources of type `Connection`, `ResultSet`, and `Statement`.

## Example: Java Database Connectivity with MySQL

To connect Java application with the MySQL database, we need to follow 5 following steps.

In this example we are using MySQL as the database. So we need to know following information for the mysql database:

**Step1.Driver class:** The driver class for the mysql database is `com.mysql.jdbc.Driver`.

**Step2.Connection URL:** The connection URL for the mysql database is `jdbc:mysql://localhost:3306/sonoo` where `jdbc` is the API, `mysql` is the database, `localhost` is the server name on which mysql is running, we may also use IP address, `3306` is the port number and `sonoo` is the database name. We may use any database, in such case, we need to replace the `sonoo` with our database name.

**Step3.Username:** The default username for the mysql database is `root`.

**Step4.Password:** It is the password given by the user at the time of installing the mysql database. In this example, we are going to use `root` as the password.

**Step:5 JDBC imports:**

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.SQLException;  
import java.sql.ResultSet;  
import java.sql.Statement;
```

Each of these imports provides access to a class that facilitates the standard Java database connection:

**Connection** represents the connection to the database.

**DriverManager** obtains the connection to the database. (Another option is `DataSource`, used for connection pooling.)

**SQLException** handles SQL errors between the Java application and the database.

**ResultSet** and **Statement** model the data result sets and SQL statements.

## Connect Java Application with mysql database

### AIM:

Create a Java JDBC program to access the Mysql database

### Procedure:

Step 1: Start the Netbeans IDE

Step 2: File -> New project -> java -> application -> Project Name -> Next

Step 3: Project name -> right click -> java -> java class -> Finish

Step 4: Create a appropriate JDBC java code to connection with Mysql

Step 5: Create a database and table for student of the S\_id, Sname, DOB, Address and Email\_id in Mysql

Step 6: Execute the Mysql database in JDBC java program.

Step 7: Stop

### Mysql database:

```
mysql> use college
```

```
Database changed
```

```
mysql> desc mca;
```

```
Mysql> create table mca(S_id int(5)primary key,Sname varchar(20),DOB date,Address  
varchar(20),Email_id varchar(20));
```

```
Mysql> insert into mca values(1001,"Raja",'2023-07-09',"Chennai","ss@gmail.com");
```

```
Mysql> insert into mca values(1001,"John",'2023-07-10',"Mangalore","vv@gmail.com");
```

```
mysql> select * from mca;
```

```
+-----+-----+-----+-----+-----+  
| S_id | Sname | DOB      | Address | Email_id |  
+-----+-----+-----+-----+-----+  
| 1001 | Raja  | 2023-07-09 | Chennai | ss@gmail.com |  
| 1002 | John  | 2023-07-10 | Mangalore | vv@yahoo.com |  
+-----+-----+-----+-----+-----+
```

```
2 rows in set (0.00 sec)
```

### **Mysql JDBC program**

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class domo {

    public static void main(String args[]) {
        try {
            Connection con = (Connection)
DriverManager.getConnection("jdbc:mysql://localhost:3306/college", "root", "root");
            Statement stnt = con.createStatement();
            String query = "select*from mca";
            ResultSet rs = stnt.executeQuery(query);
            while (rs.next()) {
                for (int i=1;i<=5;i++){

                    System.out.print(rs.getString(i));
                    System.out.println("|");
                }
                System.out.println();
            }
        }
        catch (SQLException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

### **Output:**

```
run:
1001|
Raja|
2023-07-09|
Chennai|
ss@gmail.com|
1002|
John|
2023-07-10|
Mangalore|
vv@yahoo.com|
```

BUILD SUCCESSFUL (total time: 0 seconds)

### **Result:**

Thus program has been successfully executed.

## JDBC statements

The JDBC Statement, CallableStatement, and PreparedStatement interfaces define the methods and properties that enable to send SQL or PL/SQL commands and receive data from your database. They also define methods that help bridge data type differences between Java and SQL data types used in a database.

### Syntax:

Statement statement=connection.createStatement(); Implementation: Once the Statement object is created, there are three ways to execute it.

**Statement** : Used to implement simple SQL statements with no parameters.

**PreparedStatement** : (Extends Statement .) Used for precompiling SQL statements that might contain input parameters. ...

**CallableStatement**: (Extends PreparedStatement .)

### ResultSet:

The java. sql. ResultSet interface represents the result set of a database query. A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

## JDBC - Statements, PreparedStatement and CallableStatement

The JDBC Statement, CallableStatement, and PreparedStatement interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database. They also define methods that help bridge data type differences between Java and SQL data types used in a database.

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

## JDBC Statements

The Statement Objects: Creating Statement Object Before , can use a Statement object to execute a SQL statement, you need to create one using the Connection object's `createStatement( )` method, as in the following example –

### Syntax:

```
Statement stmt = null;
```

```
try {
```

```
    stmt = conn.createStatement( );
```

```
    ...
```

```
}
```

```
catch (SQLException e) {
```

```
    ...
```

```
}
```



```
finally {
```

```
    ...
```

```
}
```

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods. **boolean execute (String SQL):** Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.[DDL create,drop,alter]

**int executeUpdate (String SQL) –** Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.[DML select,update,delete]

**ResultSet executeQuery (String SQL) –** Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement

## **Closing Statement Object**

Just as close a Connection object to save database resources, for the same reason should also close the Statement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

### **Syntax:**

```
Statement stmt = null;
```

```
try {
```

```
    stmt = conn.createStatement( );
```

```
    ...
```

```
}
```

```
catch (SQLException e) {
```

```
    ...
```

```
}
```

```
finally {
    stmt.close();
}
```

## **JDBC PreparedStatement**

The PreparedStatement Objects:

The PreparedStatement interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

Creating PreparedStatement Object:

```
PreparedStatement pstmt = null;
```

```
try {
    String SQL = "Update Employees SET age = ? WHERE id = ?";
    pstmt = conn.prepareStatement(SQL);
    ...
}
catch (SQLException e) {
    ...
}
finally {
    ...
}
```

All parameters in JDBC are represented by the ? symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The setXXX() methods bind values to the parameters, where XXX represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

All of the Statement object's methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

### **Closing PreparedStatement Object:**

A statement just as close a Statement object, for the same reason you should also close the PreparedStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

Syntax:

```
PreparedStatement pstmt = null;
```

```
try {
```

```
    String SQL = "Update Employees SET age = ? WHERE id = ?";
```

```
    pstmt = conn.prepareStatement(SQL);
```

```
    ...
```

```
}
```

```
catch (SQLException e) {
```

```
    ...
```

```
}
```

```
finally {
    pstmt.close();
}
```

## CallableStatement

### The CallableStatement Objects:

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

Creating CallableStatement Object

Example: To execute the following Oracle stored procedure :

```
CREATE OR REPLACE PROCEDURE getEmpName
    (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS
BEGIN
    SELECT first INTO EMP_FIRST
    FROM Employees
    WHERE ID = EMP_ID;
END;
```

NOTE – Above stored procedure has been written for Oracle, but we are working with MySQL database so, let us write same stored procedure for MySQL as follows to create it in EMP database –

#### Example: CallableStatement using Mysql

```
DELIMITER $$
DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$
CREATE PROCEDURE `EMP`.`getEmpName`
    (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
    SELECT first INTO EMP_FIRST
    FROM Employees
    WHERE ID = EMP_ID;
END $$
```

DELIMITER ;

### CallableStatement

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

Here are the definitions of each. Parameter Description

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

### CallableStatement

The following code snippet shows how to employ the Connection.prepareCall() method to instantiate a CallableStatement object based on the preceding stored procedure –

```
CallableStatement cstmt = null;
```

```
try {
```

```
    String SQL = "{call getEmpName (?, ?)}";
```

```
    cstmt = conn.prepareCall (SQL);
```

```

    ...
}
catch (SQLException e) {
    ...
}
finally {
    ...
}

```

## CallableStatement

The String variable SQL, represents the stored procedure, with parameter placeholders.

Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException.

If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return.

Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

## CallableStatement

**Closing CallableStatement Object:** Just as you close other Statement object, for the same reason you should also close the CallableStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

```
CallableStatement cstmt = null;
```

```
try {
```

```

String SQL = "{call getEmpName (?, ?)}";

cstmt = conn.prepareCall (SQL);

...

}

catch (SQLException e) {

    ...

}

finally {

    cstmt.close();

}

```

## JDBC Resultset

The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The java.sql.ResultSet interface represents the result set of a database query.

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

### The methods of the ResultSet interface can be broken down into three categories –

**Navigational methods** – Used to move the cursor around.

**Get methods** – Used to view the data in the columns of the current row being pointed by the cursor.

**Update methods** – Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created.



## JDBC Resultset

JDBC provides the following connection methods to create statements with desired ResultSet –

```
createStatement(int RSType, int RSConcurrency);
```

```
prepareStatement(String SQL, int RSType, int RSConcurrency);
```

```
prepareCall(String sql, int RSType, int RSConcurrency);
```

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

## Type of ResultSet

The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE\_FORWARD\_ONLY.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

## Concurrency of ResultSet

The possible RSConcurrency are given below. If you do not specify any Concurrency type, you will automatically get one that is CONCUR\_READ\_ONLY.

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set.  This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

### Note:

All our examples written so far can be written as follows, which initializes a Statement object to create a forward-only, read only ResultSet object –

## Concurrency of Results

```
try {  
    Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,  
                                           ResultSet.CONCUR_READ_ONLY);  
}  
catch(Exception ex) {  
    ....  
}  
finally {  
    ....  
}
```

## JDBC Resultset:

**Example:** The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The java.sql.ResultSet interface represents the result set of a database query.

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories –

- Navigational methods – Used to move the cursor around.
- Get methods – Used to view the data in the columns of the current row being pointed by the cursor.
- Update methods – Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created.

JDBC provides the following connection methods to create statements with desired ResultSet –

- `createStatement(int RSType, int RSConcurrency);`
- `prepareStatement(String SQL, int RSType, int RSConcurrency);`
- `prepareCall(String sql, int RSType, int RSConcurrency);`

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

## **Navigating a Result Set**

There are several methods in the ResultSet interface that involve moving the cursor, including –

### **S.N. Methods & Description**

- |   |  |
|---|--|
| 1 | <b><code>public void beforeFirst() throws SQLException</code></b><br>Moves the cursor just before the first row. |
| 2 | <b><code>public void afterLast() throws SQLException</code></b><br>Moves the cursor just after the last row.     |
| 3 | <b><code>public boolean first() throws SQLException</code></b><br>Moves the cursor to the first row.             |
| 4 | <b><code>public void last() throws SQLException</code></b><br>Moves the cursor to the last row.                  |
| 5 | <b><code>public boolean absolute(int row) throws SQLException</code></b>   |

Moves the cursor to the specified row.

- 6     **public boolean relative(int row) throws SQLException**  
     Moves the cursor the given number of rows forward or backward, from where it is currently pointing.

- 7     **public boolean previous() throws SQLException**  
     Moves the cursor to the previous row. This method returns false if the previous row is off the result set.

- 8     **public boolean next() throws SQLException**  
     Moves the cursor to the next row. This method returns false if there are no more rows in the result set.

- 9     **public int getRow() throws SQLException**  
     Returns the row number that the cursor is pointing to.

- 10    **public void moveToInsertRow() throws SQLException**  
     Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered.

- 11    **public void moveToCurrentRow() throws SQLException**  
     Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing

## Viewing a Result Set

The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions –

- One that takes in a column name.
- One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet

–

## S.N. Methods & Description

- |   |   |
|---|---|
|   | <b>public int getInt(String columnName) throws SQLException</b>   |
| 1 | Returns the int in the current row in the column named columnName.  |
|   | <b>public int getInt(int columnIndex) throws SQLException</b>   |
| 2 | Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on. |

Similarly, there are get methods in the ResultSet interface for each of the eight Java primitive types, as well as common types such as java.lang.String, java.lang.Object, and java.net.URL.

There are also methods for getting SQL data types java.sql.Date, java.sql.Time, java.sql.Timestamp, java.sql.Clob, and java.sql.Blob. Check the documentation for more information about using these SQL data types.

## Updating a Result Set

The ResultSet interface contains a collection of update methods for updating the data of a result set.

As with the get methods, there are two update methods for each data type –

- One that takes in a column name.
- One that takes in a column index.

For example, to update a String column of the current row of a result set, you would use one of the following updateString() methods –

## S.N. Methods & Description

- |   |  |
|---|--|
|   | <b>public void updateString(int columnIndex, String s) throws SQLException</b> |
| 1 | Changes the String in the specified column to the value of s.                  |

- public void updateString(String columnName, String s) throws SQLException**
- 2 Similar to the previous method, except that the column is specified by its name instead of its index.

There are update methods for the eight primitive data types, as well as String, Object, URL, and the SQL data types in the java.sql package.

Updating a row in the result set changes the columns of the current row in the ResultSet object, but not in the underlying database. To update your changes to the row in the database, you need to invoke one of the following methods.

#### **S.N. Methods & Description**

- public void updateRow()**
- 1 Updates the current row by updating the corresponding row in the database.
- public void deleteRow()**
- 2 Deletes the current row from the database
- public void refreshRow()**
- 3 Refreshes the data in the result set to reflect any recent changes in the database.
- public void cancelRowUpdates()**
- 4 Cancels any updates made on the current row.
- public void insertRow()**
- 5 Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.

### **JDBC - Data Types**

The JDBC driver converts the Java data type to the appropriate JDBC type, before sending it to the database. It uses a default mapping for most data types. For example, a Java int is converted to an SQL INTEGER. Default mappings were created to provide consistency between drivers.

The following table summarizes the default JDBC data type that the Java data type is converted to, when you call the setXXX() method of the PreparedStatement or CallableStatement object or the ResultSet.updateXXX() method.

S.NO	SQL	JDBC/Java	setXXX	updateXXX
1	VARCHAR	java.lang.String	setString	updateString
2	CHAR	java.lang.String	setString	updateString
3	LONGVARCHAR	java.lang.String	setString	updateString
4	BIT	boolean	setBoolean	updateBoolean
5	NUMERIC	java.math.BigDecimal	setBigDecimal	updateBigDecimal
6	TINYINT	byte	setByte	updateByte
7	SMALLINT	short	setShort	updateShort
8	INTEGER	int	setInt	updateInt
9	BIGINT	long	setLong	updateLong
10	REAL	float	setFloat	updateFloat
11	FLOAT	float	setFloat	updateFloat



## JDBC data types

S.NO	SQL	JDBC/Java	setXXX	updateXXX
12	DOUBLE	double	setDouble	updateDouble
13	VARBINARY	byte[]	setBytes	updateBytes
14	BINARY	byte[]	setBytes	updateBytes
15	DATE	java.sql.Date	setDate	updateDate
15	TIME	java.sql.Time	setTime	updateTime
17	TIMESTAMP	java.sql.Timestamp	setTimestamp	updateTimestamp
18	CLOB	java.sql.Clob	setClob	updateClob
19	BLOB	java.sql.Blob	setBlob	updateBlob
20	ARRAY	java.sql.Array	setARRAY	updateARRAY
21	REF	java.sql.Ref	SetRef	updateRef
22	STRUCT	java.sql.Struct	SetStruct	updateStruct

JDBC 3.0 has enhanced support for BLOB, CLOB, ARRAY, and REF data types. The ResultSet object now has updateBLOB(), updateCLOB(), updateArray(), and updateRef() methods that enable you to directly manipulate the respective data on the server.

The setXXX() and updateXXX() methods enable you to convert specific Java types to specific JDBC data types. The methods, setObject() and updateObject(), enable you to map almost any Java type to a JDBC data type.

ResultSet object provides corresponding getXXX() method for each data type to retrieve column value. Each method can be used with column name or by its ordinal position.

# JDBC data types

S.NO	SQL	JDBC/Java	setXXX	updateXXX
1	VARCHAR	java.lang.String	setString	getString
2	CHAR	java.lang.String	setString	getString
3	LONGVARCHAR	java.lang.String	setString	getString
4	BIT	boolean	setBoolean	getBoolean
5	NUMERIC	java.math.BigDecimal	setBigDecimal	getBigDecimal
6	TINYINT	byte	setByte	getByte
7	SMALLINT	short	setShort	getShort
8	INTEGER	int	setInt	getInt
9	BIGINT	long	setLong	getLong
10	REAL	float	setFloat	getFloat
11	FLOAT	float	setFloat	getFloat

## JDBC data types

S.NO	SQL	JDBC/Java	setXXX	updateXXX
12	DOUBLE	double	setDouble	getDouble
13	VARBINARY	byte[]	setBytes	getBytes
14	BINARY	byte[]	setBytes	getBytes
15	DATE	java.sql.Date	setDate	getDate
16	TIME	java.sql.Time	setTime	getTime
17	TIMESTAMP	java.sql.Timestamp	setTimestamp	getTimestamp
18	CLOB	java.sql.Clob	setClob	getClob
19	BLOB	java.sql.Blob	setBlob	getBlob
20	ARRAY	java.sql.Array	setARRAY	getARRAY
21	REF	java.sql.Ref	SetRef	getRef
22	STRUCT	java.sql.Struct	SetStruct	getStruct

### Example: Date & Time Data Types

The `java.sql.Date` class maps to the SQL DATE type, and the `java.sql.Time` and `java.sql.Timestamp` classes map to the SQL TIME and SQL TIMESTAMP data types, respectively.

Following example shows how the Date and Time classes format the standard Java date and time values to match the SQL data type requirements.

```
import java.sql.Date;
import java.sql.Time;
import java.sql.Timestamp;
import java.util.*;

public class SqlDateTime {
    public static void main(String[] args) {
        //Get standard date and time
        java.util.Date javaDate = new java.util.Date();
        long javaTime = javaDate.getTime();
        System.out.println("The Java Date is:" +
            javaDate.toString());

        //Get and display SQL DATE
        java.sql.Date sqlDate = new
java.sql.Date(javaTime);
        System.out.println("The SQL DATE is: " +
```

```

        sqlDate.toString());

    //Get and display SQL TIME
    java.sql.Time sqlTime = new
java.sql.Time(javaTime);
    System.out.println("The SQL TIME is: " +
        sqlTime.toString());
    //Get and display SQL TIMESTAMP
    java.sql.Timestamp sqlTimestamp =
new java.sql.Timestamp(javaTime);
    System.out.println("The SQL TIMESTAMP is: " +
        sqlTimestamp.toString());
} //end main
} //end SqlDateTime

```

Output:

The Java Date is: Tue Aug 18 13:46:02 GMT+04:00 2009

The SQL DATE is: 2009-08-18

The SQL TIME is: 13:46:02

The SQL TIMESTAMP is: 2009-08-18 13:46:02.828

## Transactions in JDBC

A SQL transaction is a grouping of one or more SQL statements that interact with a database. A transaction in its entirety can commit to a database as a single logical unit or rollback (become undone) as a single logical unit. In SQL, transactions are essential for maintaining database integrity

### Things required for transaction in JDBC:

To do transaction management in Jdbc, we need to follow the below steps.

Step 1: Disable auto commit mode of Jdbc

Step 2: Put all operation of a transaction in try block.

Step 3: If all operation are done successfully then commit in try block, otherwise rollback in catch block.

By default in Jdbc autocommit mode is enabled but we need to disable it. To disable call `setAutoCommit()` method of connection Interface.

# Transactions

## Example

- `con.setAutoCommit(false);`
- To commit a transaction, call `commit()` and to rollback a transaction, call `rollback()` method of connection Interface respectively.

## Example

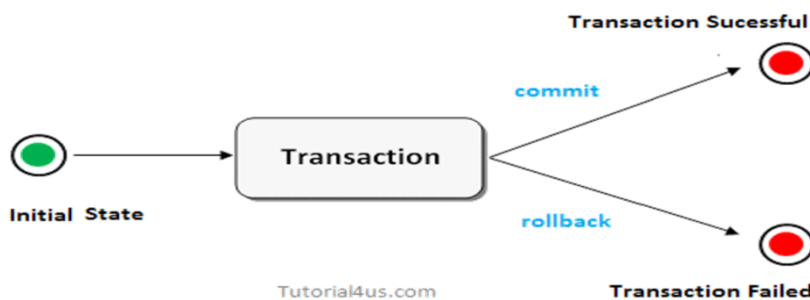
- `con.commit();`
- `con.rollback();`

**Note:** In transaction management DDL create ,drop and alter operation are not allowed.

**Note:** The operation in a transaction management may be executed on same table or different table but database should be same.

## Transaction Management in JDBC

A transaction is a group of operation used to performed one task if all operations in the group are success then the task is finished and the transaction is successfully completed. If any one operation in the group is failed then the task is failed and the transaction is failed.



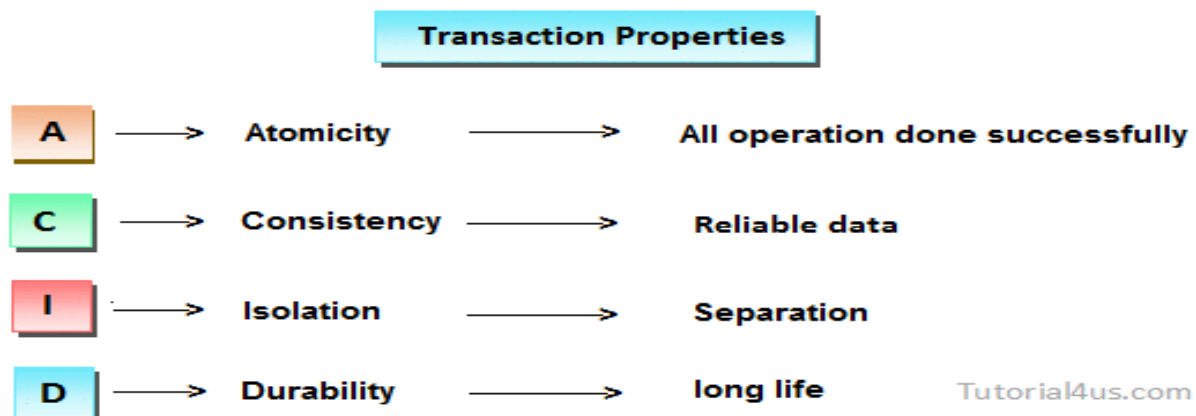
# Ticket Reservation transaction

Suppose a Movie or Travel ticket booking at online is a transaction. This task contains four operation.

- Verify the seats
- Reserve the seats
- Payment
- Issue tickets
- If all the above four operations are done successfully then a transaction is finished successfully. In the middle, if any one operation is failed then all operation are canceled and finally a transaction is failed.

## Properties of Transaction managements

Every transaction follows some transaction properties these are called ACID properties



## ACID

**Atomicity:** Atomicity of a transaction is nothing but in a transaction either all operations can be done or all operation can be undone, but some operations are done and some operation are undone should not occur.

**Consistency:** Consistency means, after a transaction completed with successful, the data in the data store should be a reliable data this reliable data is also called as consistent data.

**Isolation:** Isolation means, if two transaction are going on same data then one transaction will not disturb another transaction.

**Durability:** Durability means, after a transaction is completed the data in the data store will be permanent until another transaction is going to be performed on that data.

## Advantage of Transaction Management

fast performance It makes the performance fast because database is hit at the time of commit.

- **Types of Transaction**
- Local Transaction
- Distributed or global transaction
- **Local Transaction**
- A local transaction means, all operation in a transaction are executed against one database.

For example;

If transfer money from first account to second account belongs to same bank then transaction is local transaction.

- **Global Transaction**

- A global transaction means, all operations in a transaction are executed against multiple database.

**For Example;**

If transfer money from first account to second account belongs to different banks then the transaction is a global transaction.

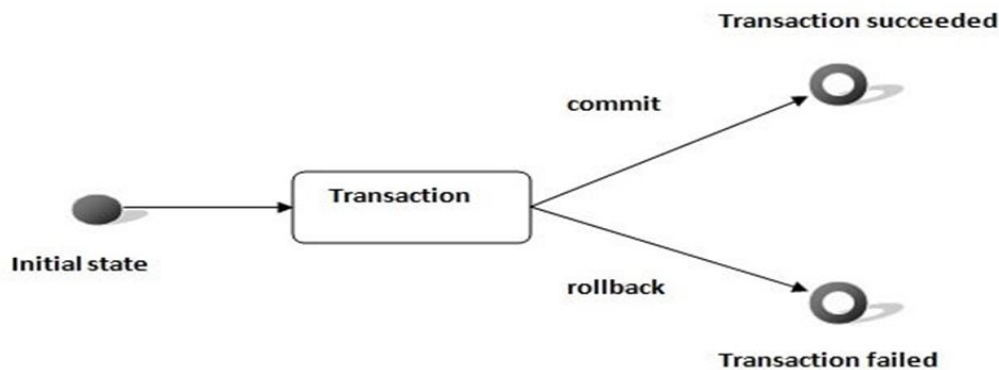
- **Note:** Jdbc technology perform only local transactions. For global transaction in java we need either EJB or spring framework.

### Useful Connection Methods (for Transactions)

- **getAutoCommit/setAutoCommit**
  - By default, a connection is set to auto-commit
  - Retrieves or sets the auto-commit mode
- **commit**
  - Force all changes since the last call to commit to become permanent
  - Any database locks currently held by this Connection object are released
- **rollback**
  - Drops all changes since the previous call to commit
  - Releases any database locks held by this Connection object

Advantage of Transaction Mangement

Fast performance It makes the performance fast because database is hit at the time of commit.



In JDBC, Connection interface provides methods to manage transaction

Method	Description
void setAutoCommit(boolean status)	It is true by default means each transaction is committed by default.
void commit()	commits the transaction.
void rollback()	cancels the transaction.

### **Example of transaction management in jdbc using Statement**

```
import java.sql.*;
class FetchRecords{
public static void main(String args[])throws Exception{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle"
);
con.setAutoCommit(false);
Statement stmt=con.createStatement();
stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");
con.commit();
con.close();
}} // If you see the table emp400, you will see that 2 records has been added.
```

### **Example of transaction management using PreparedStatement**

```
import java.sql.*;
import java.io.*;
class TM{
public static void main(String args[]){
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:152
1:xe","system","oracle");
con.setAutoCommit(false);
PreparedStatement ps=con.prepareStatement("insert into user420
values(?,?,?)");
BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
while(true){
System.out.println("enter id");
String s1=br.readLine();
int id=Integer.parseInt(s1);
System.out.println("enter name");
String name=br.readLine();
```



```

System.out.println("enter salary");
String s3=br.readLine();
int salary=Integer.parseInt(s3);
ps.setInt(1,id);
ps.setString(2,name);
ps.setInt(3,salary);
ps.executeUpdate();
System.out.println("commit/rollback");
String answer=br.readLine();
if(answer.equals("commit")){
con.commit();
}
if(answer.equals("rollback")){
con.rollback();
}

```

```

System.out.println("Want to add more records y/n");
String ans=br.readLine();
if(ans.equals("n")){
break;
}

}
con.commit();
System.out.println("record successfully saved");
con.close();//before closing connection commit() is called
}catch(Exception e){System.out.println(e);}

```

}} //It will ask to add more records until you press n. If you press n, transaction is committed.

## Example: SQL Query language

### COMMIT:

COMMIT in SQL is a transaction control language that is used to permanently save the changes done in the transaction in tables/databases. The database cannot regain its previous state after its execution of commit.

Example: Consider the following Staff table with records: STAFF

Id	Name	Age	Allowance	Salary
1	Rahul	26	400	4000
2	Khaitan	46	900	9000
3	Munjai	36	400	4500
4	Ram	28	800	8000
5	Manav	24	400	6500
6	Kaira	21	700	7800

```
sql>  
SELECT *  
FROM Staff  
WHERE Allowance = 400;
```

```
sql> COMMIT;
```

Id	Name	Age	Allowance	Salary
1	Rahul	26	400	4000
3	Munjal	36	400	4500
5	Manav	24	400	6500

sql>

SELECT \*

FROM Staff

WHERE Allowance = 400;

sql> COMMIT;

So, the SELECT statement produced the output consisting of three rows.

Sql>select \* from Staff;

### **Rollback**

So, the SELECT statement produced the same output with the ROLLBACK command

Id	Name	Age	Allowance	Salary
1	Rahul	26	400	4000
2	Khaitan	46	900	9000
3	Munjai	36	400	4500
4	Ram	28	800	8000
5	Manav	24	400	6500
6	Kaira	21	700	7800

## Batch Processing in JDBC

Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast. It is because when one sends multiple statements of SQL at once to the database, the communication overhead is reduced significantly, as one is not communicating with the database frequently, which in turn results to fast performance.

The `java.sql.Statement` and `java.sql.PreparedStatement` interfaces provide methods for batch processing.

### Advantage of Batch Processing

Fast Performance

## Batch Processing

Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database. When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.

### Note:

JDBC drivers are not required to support this feature.

### Methods of Statement interface

The required methods for batch processing are given below:

Method	Description
<code>void addBatch(String query)</code>	The <code>addBatch(String query)</code> method of the <code>CallableStatement</code> , <code>PreparedStatement</code> , and <code>Statement</code> is used to single statements to a batch.
<code>int[] executeBatch()</code>	The <code>executeBatch()</code> method begins the execution of all the grouped together statements. The method returns an integer array, and each of the element of the array represents the updated count for respective update statement.
<code>boolean DatabaseMetaData.supportsBatchUpdates() throws SQLException</code>	If the target database facilitates the batch update processing, then the method returns true.
<code>void clearBatch()</code>	The method removes all the statements that one has added using the <code>addBatch()</code> method.

## Example of batch processing in JDBC

Simple example of batch processing in JDBC. It follows following steps:

1. Load the driver class
2. Create Connection
3. Create Statement
4. Add query in the batch
5. Execute Batch
6. Close Connection

### **1.Load the driver class**

`Class.forName()`

An efficient way to load the JDBC driver is to invoke the `Class.forName().newInstance()` method, specifying the name of the driver class, as in the following example: `Class`.

The class loading process triggers a static initialization routine that registers the driver instance with the `DriverManager` and associates this class with the database engine identifier, such as `oracle` or `postgres`. After the registration is complete, we can use this identifier inside the JDBC URL as `jdbc:oracle`.

### **2.Create Connection**

Create the connection object

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

JDBC makes it possible to establish a connection with a data source, send queries and update statements, and process the results . Simply, JDBC makes it possible to do the following things within a Java application: Establish a connection with a data source. Send queries and update statements to the data source.

The JDBC Connection class, `java.sql.Connection`, represents a database connection to a relational database. Before you can read or write data from and to a database via JDBC, you need to open a connection to the database.

### **3.Create a Statement**

**Create a Statement:** From the connection interface, you can create the object for this interface. It is generally used for general-purpose access to databases and is useful while using static SQL statements at runtime. Syntax: `Statement statement = connection.`

The JDBC Statement, `CallableStatement`, and `PreparedStatement` interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database. They also define methods that help bridge data type differences between Java and SQL data types used in a database.

### **4.Add query in the batch**

The `addBatch()` method of `Statement`, `PreparedStatement`, and `CallableStatement` is used to add individual statements to the batch.

The `executeBatch()` is used to start the execution of all the statements grouped together.

Batch is a group of SQL statements that are executed at one time by SQL Server. These statements are sent to SQL Server by a program, such as the Query Analyzer. The opposite of a batch query is a single query, containing only one SQL statement.

### **5 and 6 Execute Batch and Close Connection**

`int[] executeBatch()` The `executeBatch()` method begins the execution of all the statements grouped together. The method returns an integer array, and each element of the array represents the updated count for the respective update statement.

## Batch processing: Progam

FileName:FetchRecords.java

```
import java.sql.*;
class FetchRecords{
public static void main(String args[])throws Exception{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:152
1:xe","system","oracle");
con.setAutoCommit(false);
Statement stmt=con.createStatement();
stmt.addBatch("insert into user420 values(190,'abhi',40000)");
stmt.addBatch("insert into user420 values(191,'umesh',50000)");
stmt.executeBatch();//executing the batch
con.commit();
con.close();
}} //If you see the table user420, two records have been added.
```

## Example of batch processing

using PreparedStatement

```
import java.sql.*;
import java.io.*;
class BP{
public static void main(String args[]){
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","ora
cle");
PreparedStatement ps=con.prepareStatement("insert into user420 values(?,?,?)");
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
while(true){
System.out.println("enter id");
String s1=br.readLine();
int id=Integer.parseInt(s1);
System.out.println("enter name");
String name=br.readLine();
System.out.println("enter salary");
String s3=br.readLine();
int salary=Integer.parseInt(s3);
ps.setInt(1,id);
ps.setString(2,name);
ps.setInt(3,salary);
```



```

ps.addBatch();
System.out.println("Want to add more records y/n");
String ans=br.readLine();
if(ans.equals("n")){
break;
}
}
ps.executeBatch();// for executing the batch
System.out.println("record successfully saved");
con.close();
}catch(Exception e){System.out.println(e);}
}}

```

```

enter id
101
enter name
Manoj Kumar
enter salary
10000
Want to add more records y/n
y
enter id
101
enter name
Harish Singh
enter salary
15000
Want to add more records y/n
y
enter id
103
enter name
Rohit Anuragi
enter salary
30000
Want to add more records y/n
y
enter id
104
enter name
Amrit Gautam
enter salary
40000
Want to add more records y/n
n
record successfully saved

```

It will add the queries into the batch until user press n. Finally, it executes the batch. Thus, all the added queries will be fired.

# Stored procedures

Stored procedures are sub routines, segment of SQL statements which are stored in SQL catalog. All the applications that can access Relational databases (Java, Python, PHP etc.), can access stored procedures. Stored procedures contain IN and OUT parameters or both. A stored procedure is a set of Structured Query Language (SQL) statements with an assigned name, which are stored in a relational database management system (RDBMS) as a group, so it can be reused and shared by multiple programs. To call stored procedures, you invoke methods in the CallableStatement or PreparedStatement class. The basic steps for calling a stored procedures using standard CallableStatement methods are: Invoke the Connection. prepareCall method with the CALL statement as its argument to create a CallableStatement object.

## Uses:

By grouping SQL statements, a stored procedure allows them to be executed with a single call. This minimizes the use of slow networks, reduces network traffic, and improves round-trip response time. OLTP applications, in particular, benefit because result set processing eliminates network bottlenecks.

## Stored procedure:

Group of SQL statements that form a logical unit and perform a particular task. This information is made available through a DatabaseMetaData object.

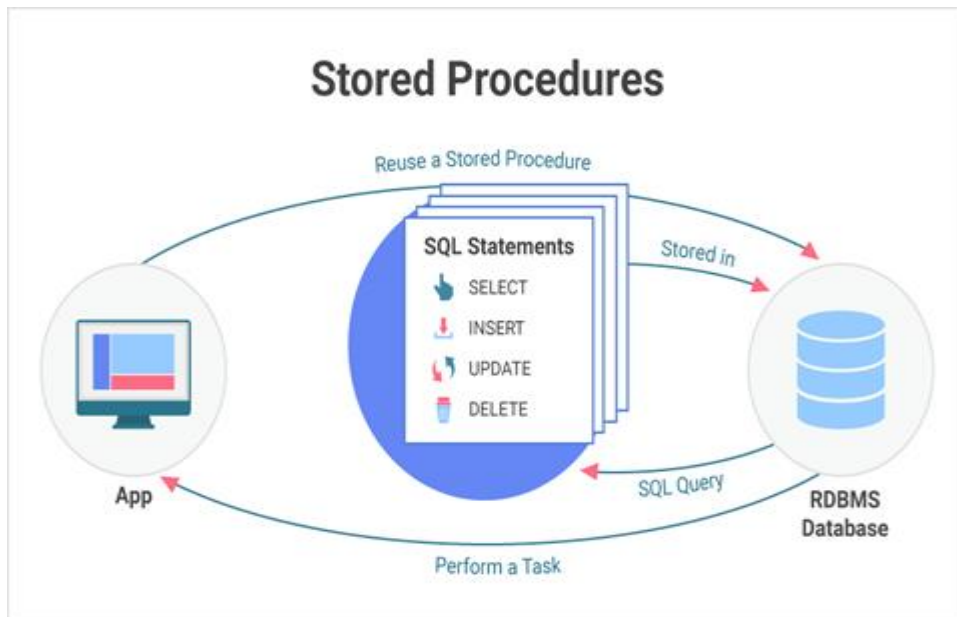
## Stored procedure from JDBC

To call stored procedures, you invoke methods in the CallableStatement or PreparedStatement class. The basic steps for calling a stored procedures using standard CallableStatement methods are: Invoke the Connection. prepareCall method with the CALL statement as its argument to create a CallableStatement object.

## Stored Procedure in SQL

A stored procedure is a set of Structured Query Language (SQL) statements with an assigned name, which are stored in a relational database management system (RDBMS) as a group, so it can be reused and shared by multiple programs. The function must return a value but in Stored Procedure it is optional. Even a procedure can return zero or n values. Functions can have only input parameters for it whereas Procedures can have input or output parameters. Functions can be called from Procedure whereas Procedures cannot be called from a Function

CallableStatement in JDBC is an interface present in a java.sql package and it is the child interface of Prepared Statement. Callable Statement is used to execute the Stored Procedure and functions. The driver software vendor is responsible for providing the implementations for Callable Statement interface.



A stored procedure is a set of SQL statements and other PL/SQL constructs stored in a relational database management system (RDBMS) as a group that you can save and reuse repeatedly. A stored procedure can consist of multiple SQL statements like SELECT, INSERT, UPDATE, or DELETE. They run as a unit and are used to solve a specific problem or perform a set of related tasks. That means that if you need an SQL query and write it repeatedly, you can save it as a stored procedure and then call it to perform the query. Using stored procedures can simplify and accelerate the execution of SQL queries.

**For example: storing procedures can reduce network traffic between servers and clients**

**Note: Example**

Call the stored procedure getempinfo that receives employee no: as an input and provides corresponding name and salary of the employee. JDBC

```
import java.sql.*;
class EmployeeInfo
{
    public static void main(String args[]) throws Exception
    {
        Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/emp", "root",
        "");
        CallableStatement cst = con.prepareCall("{call getempinfo( ? ? ? )}");
        cst.setInt(1,100);
        cst.registerOutParameter(2,Types.varchar);
        cst.registerOutParameter(3,Types.Float);
        cst.execute();

        System.out.println("Employee name is" , +cst.getString(2));
        System.out.println("Employee salary is" , +cst.getFloat(3));
    }
}
```

**Example:**

A Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

**Creating CallableStatement Object**

Suppose, you need to execute the following Oracle stored procedure –

```
CREATE OR REPLACE PROCEDURE getEmpName
  (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS
BEGIN
  SELECT first INTO EMP_FIRST
  FROM Employees
  WHERE ID = EMP_ID;
END;
```

**Example: How to Use Callable Statement in Java to Call Stored Procedure?**

The CallableStatement of JDBC API is used to call a stored procedure. A Callable statement can have output parameters, input parameters, or both. The prepareCall() method of connection interface will be used to create CallableStatement object.

Following are the steps to use Callable Statement in Java to call Stored Procedure:

**1) Load MySQL driver and Create a database connection.**

```
import java.sql.*;
public class JavaApplication1 {

    public static void main(String[] args) throws Exception

    {

        Class.forName("com.mysql.jdbc.Driver");

        Connection
con=DriverManager.getConnection("jdbc:mysql://localhost/root","geek","geek");

    }
}
```

**2) Create a SQL String**

We need to store the SQL query in a String.

```
String sql_string="insert into student values(?,?,?)";
```

### 3) Create CallableStatement Object

The `prepareCall()` method of connection interface will be used to create `CallableStatement` object. The `sql_string` will be passed as an argument to the `prepareCall()` method.

```
CallableStatement cs = con.prepareCall(sql_string);
```

### 4) Set The Input Parameters

Depending upon the data type of query parameters we can set the input parameter by calling `setInt()` or `setString()` methods.

```
cs.setString(1,"geek1");
```

```
cs.setString(2,"python");
```

```
cs.setString(3,"beginner");
```

### 5) Call Stored Procedure

Execute stored procedure by calling `execute()` method of `CallableStatement` class.

**Example of using Callable Statement in Java to call Stored Procedure**

```
//Java program to use Callable Statement
// in Java to call Stored Procedure

package javaapplication1;

import java.sql.*;

public class JavaApplication1 {

    public static void main(String[] args) throws Exception
    {
        Class.forName("com.mysql.jdbc.Driver");

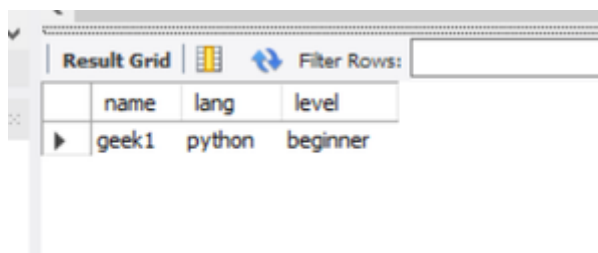
        // Getting the connection
        Connection con = DriverManager.getConnection("jdbc:mysql://localhost/root", "acm", "acm");

        String sql_string = "insert into students values(?,?,?)";

        // Preparing a CallableStatement
        CallableStatement cs = con.prepareCall(sql_string);

        cs.setString(1, "geek1");
        cs.setString(2, "python");
        cs.setString(3, "beginner");
        cs.execute();
        System.out.print("uploaded successfully\n");
    }
}
```

## Output:



	name	lang	level
▶	geek1	python	beginner

*students table after running code*