

Advanced Java

UNIT-2

Syllabus: The JDBC Connectivity Model, Database Programming :Connecting to the Database, Creating a SQL Query, Getting the Results, Updating Database Data, Error Checking and the SQL Exception Class, The SQL Warning Class, The Statement Interface, PreparedStatement, CallableStatement The ResultSet Interface, Updatable Result Sets, JDBC Types, Executing SQL Queries, Result Set Meta Data, Executing SQL Updates, Transaction Management.

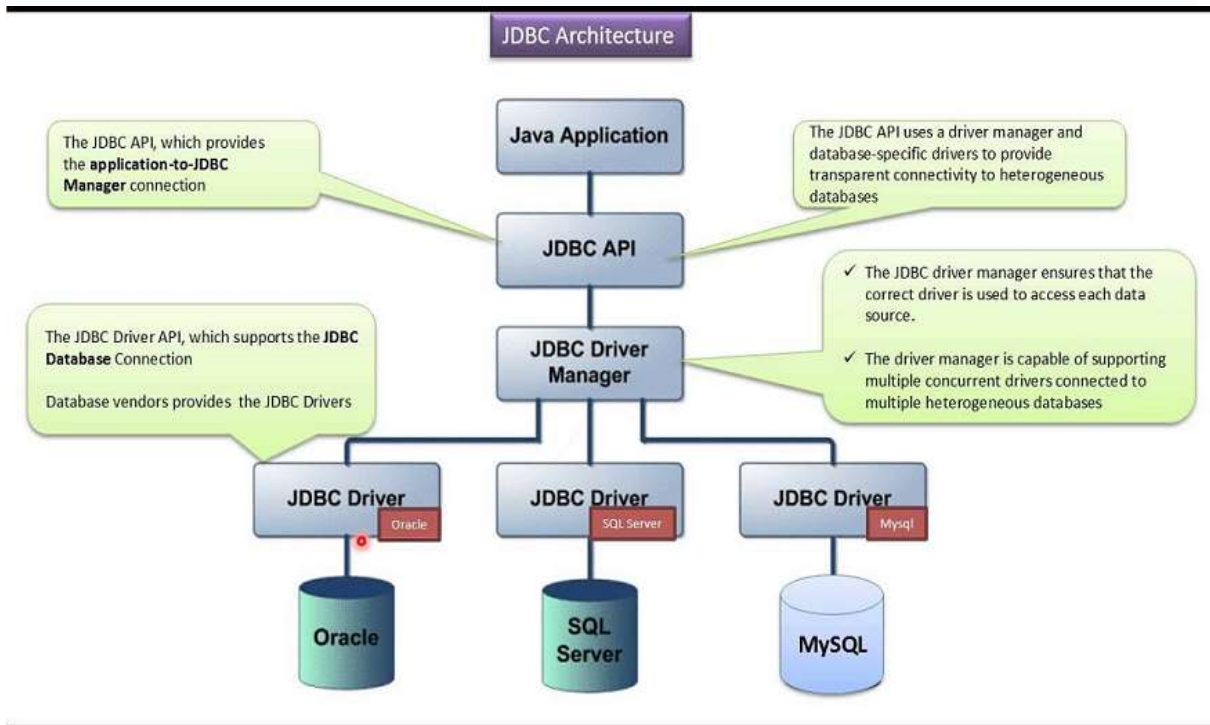
The JDBC Connectivity Model:(JDBC Architecture)

JDBC stands for **Java Database Connectivity** is a Java API (Application Programming Interface) used to interact with databases.

It consists of classes and interfaces of JDBC that allow the applications to access databases and send requests made by users to the specified database.

JDBC is a specification from **Sun Microsystems** and it is used by Java applications to communicate with relational databases.

JDBC API helps **Java applications** interact with different databases like **MSSQL, ORACLE, DB2, etc.**



Description:

JDBC architecture is **divided into 4 main components: Application, JDBC API, DriverManager, and JDBC Drivers.**

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: DriverManager, Connection, Statement, ResultSet e.t.c.
3. **DriverManager:** It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases. The JDBC driver manager makes sure that the correct driver is being used to access the databases. It is also capable of handling multiple drivers connected to multiple databases simultaneously.
4. **JDBC drivers:** To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

The JDBC classes are contained in the Java Package **java.sql** and **javax.sql**. 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
-

JDBC Drivers

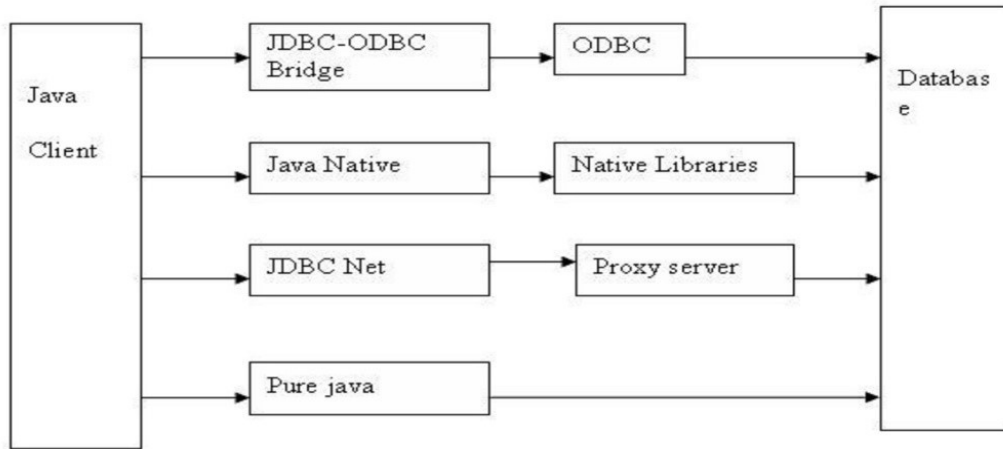
A JDBC driver is a software component that provides the facility to interact java application with the database.

A JDBC driver is a middleware layer that translates the JDBC calls to the vendor-specific APIs. 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 or Pure Java driver



1. Type-1 driver (or)JDBC-ODBC bridge driver:

JDBC-ODBC bridge driver or Type-1 driver is a native code driver which uses ODBC driver to connect with the database. It converts JDBC method calls into ODBC function calls. It is also known as Type 1 driver. Oracle has stopped its support for the JDBC-ODBC Bridge from Java 8 and above.

Advantages:

1. It is a database independent driver that is it can be used with any database for which an ODBC driver is installed.

Disadvantages:

1. Performance is not good because it converts JDBC method calls into ODBC function calls.
2. ODBC driver needs to be installed on the client machine.
3. ODBC drivers are platform-dependent drivers.
4. Type-1 driver isn't written in java, that's why it isn't a portable driver.

2.Type-2 driver (or) Native-API driver:

Native-API driver uses the client-side libraries of the database. It converts JDBC method calls into native calls (C or C++) of the database API. It is partially written in java. It is also known as Type 2 driver.

Advantages:

1. It is faster than a JDBC-ODBC bridge driver.

Disadvantages:

1. Platform dependent driver.
2. It is a database dependent driver
3. Driver needs to be installed separately in individual client machines
4. The Vendor client library needs to be installed on client machine.
5. Type-2 driver isn't written in java, that's why it isn't a portable driver

3. Type-3 driver (or) Network-Protocol driver:

Network-Protocol driver is a pure java driver which uses a middle-tier to convert JDBC calls directly or indirectly into a database specific call. Multiple types of databases can be accessed at the same time. Here all the database connectivity drivers are present in a single server, hence no need of individual client-side installation. It is also known as Type 3 or MiddleWare driver.

Advantages:

1. Platform independent.
2. Faster from Type1 and Type2 drivers.
3. It follows a three-tier communication approach.
4. Multiple types of databases can be accessed at the same time.

Disadvantages:

1. It requires database-specific coding to be done in the middle tier.

4. Type-4 driver (or) Thin driver:

Thin driver is a pure java driver which converts JDBC calls directly into the database specific calls. It is a platform independent driver. It is also known as Type 4 or Database-Protocol driver.

Advantages:

1. Platform independent.
2. Faster than all other drivers.
3. Does not require any native library and Middleware server, so no client-side or server-side installation.
4. It is fully written in Java language; hence they are portable drivers.

Disadvantages:

1. It is database dependent.
2. Multiple types of databases can't be accessed at the same time.

Which Driver to use When?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is type-4.
 - If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
 - Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
 - The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.
-

Database Programming: Connecting to the Database, Creating a SQL Query, Getting the Results, Updating Database Data

1. Import the Packages
2. Load and register drivers
3. Create a connection
4. Create a statement
5. Execute the query
6. Process the results
7. Close the connection

1. Import the Packages:

This includes uploading all the packages containing the JDBC classes, interfaces, and subclasses used during the database programming.

Syntax: `import java.sql.*`

2. Load and register the JDBC driver:

`Class.forName()` method is the most common approach to register driver class. It dynamically loads the driver class into the memory.

Syntax:

```
Class.forName("driverClassName");
```

Example:

To load or register Mysql Driver class:

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

To load or register OracleDriver class:

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

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.

3. Create connection:

The **getConnection()** method of DriverManager class is used to establish connection with the database. This method contains the database URL, username, and password as a parameter

Syntax:

```
Connection connection = DriverManager.getConnection(url, username, password)
```

Example:

To create a connection with MySQL database:

```
Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/student",  
                                           "root","admin@123");
```

To create a connection with Oracle database:

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

4. Create statement:

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

Syntax:

```
public Statement createStatement() throws SQLException
```

Example:

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

5. Execute statement:

Statement interface provides the methods to execute a statement. 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: ResultSet rs = stmt.executeQuery(selectQuery);

Example to execute query

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

6. Process the results

Now we check if `rs.next()` method is not null, then we display the details of that particular customer present in the “customer” table. `next()` function basically checks if there’s any record that satisfies the query, if no record satisfies the condition, then it returns null. Below is the sample code:

```
while(rs.next())
{
    System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

7. Close database connection:

After done with the database connection we have to close it. Use `close()` method of `Connection` interface to close database connection.

The statement and `ResultSet` objects will be closed automatically when we close the connection object.

Syntax: `connection.close();`

Example to Connect Java Application with mysql database

```
import java.sql.*;
class MysqlCon
{
    public static void main(String args[])
    {
        try
        {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/
student","root","admin@123");

            Statement stmt=con.createStatement();
            ResultSet rs=stmt.executeQuery("select * from studentinfo");
            while(rs.next())
                System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
            con.close();
        } catch(Exception e) { System.out.println(e);}
    }
}
```

Error Checking and the SQL Exception Class

- The `SQLException` class provides information on a database access error.
- `SQLException` is available in the `java.sql` package.
- **The `SQLException` class and its subtypes** provide information about errors and warnings that occur while a data source is being accessed.

Each SQLException provides several kinds of error information:

- Java Exception message - a string describing the error.
- SQLState - which identifies SQL error conditions
- An integer error code that is vendor specific.
- A chain to a next Exception- A reference to any other exceptions that also occurred

Constructors of SQLException class:

- **SQLException()**

Construct an SQLException; reason defaults to null, SQLState defaults to null and vendorCode defaults to 0.

- **SQLException(String reason)**

Construct an SQLException with a reason; SQLState defaults to null and vendorCode defaults to 0.

- **SQLException(String reason, String SQLState)**

Construct an SQLException with a reason and SQLState; vendorCode defaults to 0.

- **SQLException(String reason, String SQLState, int vendorCode)**

Construct a fully-specified SQLException

The SQLException object has the following methods for retrieving additional information about the exception:

Method Name	Description
getErrorCode()	It returns the error number
getMessage()	It returns the error message
getSQLState()	It returns the SQLState of the SQLException object. It can return null as well. For Database error, it will return XOPEN SQL State
getNextException()	It returns the next exception in the exception chain.
printStackTrace()	It prints the current exception and its backtrace to a standard error stream
setNextException (SQLException ex)	It is used to add another SQL exception in the chain

Example:

```
import java.sql.*;
class MysqlCon
{
    public static void main(String args[])
    {
        try
        {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/
```



```

        student","root","admin@123");

        Statement stmt=con.createStatement();
        stmt.executeUpdate("delete * from studentinfo where id=1001");
        con.close();
    }
    catch(SQLException e)
    {
        System.out.println(e.getMessage());
        System.out.println(e.getMessage());
    }
}
}

```

The SQL Warning Class:

- The SQLWarning class **provides information on a database access warnings.**
- SQLWarning is a subclass of SQLException that holds database access warnings.
- Warnings do not stop the execution of a specific application, as exceptions do.
- Warnings are silently chained to the object whose method caused it to be reported.
- Warnings may be retrieved from Connection , Statement , and ResultSet objects.

Constructors:

SQLWarning()

Construct an SQLWarning ; reason defaults to null, SQLState defaults to null and vendorCode defaults to 0.

SQLWarning(String reason)

Construct an SQLWarning with a reason(a description of warning); SQLState defaults to null and vendorCode defaults to 0.

SQLWarning(String reason, String SQLState)

Construct an SQLWarning with a reason and SQLState; vendorCode defaults to 0.

SQLWarning(String reason, String SQLState, int vendorCode)

Construct a fully specified SQLWarning.

Methods:

getErrorCode()	It returns the error number
getMessage()	It returns the error message
getSQLState()	It returns the SQLState of the SQLException object. It can return null as well. For Database error, it will return XOPEN SQL State

getNextWarning()	Get the warning chained to this one. This method returns the next SQLException in the chain, null if none.
setNextWarning(SQLWarning)	Add an SQLWarning to the end of the chain.

Example:

```
SQLWarning warning = stmt.getWarnings();
while (warning != null)
{
    System.out.println("Message: " + warning.getMessage());
    System.out.println("SQLState: " + warning.getSQLState());
    System.out.println("Vendor error code: " + warning.getErrorCode());
    warning = warning.getNextWarning();
}
```

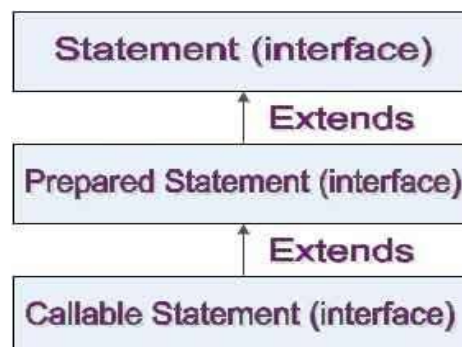
What is Jdbc Statement?

A [JDBC](#) statement object is used to execute an SQL statement. [JDBC](#) API supports three types of JDBC statement object to work with the SQL statements.

In JDBC we get three types of statements:

1. Statement Interface
2. PreparedStatement Interface
3. CallableStatement Interface

There is a relationship between above three statements:



The above three Statement Interfaces are given in [java.sql](#) package.

PreparedStatement Interface extends *Statement*. It means *PreparedStatement* interface has more features than that statement interface, *Callablestatement* (interface) extends *PreparedStatement*.

Statement: It executes normal SQL statements with no IN and OUT parameters.

Prepared Statement: It executes parameterized SQL statement that supports IN parameter.

CallableStatement: It executes parameterized SQL statement that invokes [database](#) procedure or function or supports IN and OUT parameter.

Statement interface

The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

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.(addBatch() used to add commands).

Example of Statement interface

```
import java.sql.*;
class FetchRecord{
public static void main(String args[])throws Exception{
    Class.forName("com.mysql.cj.jdbc.Driver");
    Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/
                                                Employee", "root","admin@123");

    Statement stmt=con.createStatement();

    //stmt.executeUpdate("insert into emp values(33,'Irfan',50000)");
    //int result=stmt.executeUpdate("update emp set name='Vimal',salary=10000 where id=33
    ");
    int result=stmt.executeUpdate("delete from emp765 where id=33");
    System.out.println(result+" records affected");
    con.close();
}}
```

PreparedStatement interface:

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

Let's see the example of parameterized query:

```
String sql="insert into emp values(?,?,?);"
```

As you can see, we are passing parameter (?) for the values. Its value will be set by calling the setter methods of PreparedStatement.

Why use PreparedStatement?

Improves performance: The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

How to get the instance of PreparedStatement?

The prepareStatement() method of Connection interface is used to return the object of PreparedStatement.

Syntax:

```
public PreparedStatement prepareStatement(String query)throws SQLException{}
```

Methods of PreparedStatement interface

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	sets the String value to the given parameter index.
public void setFloat(int paramIndex, float value)	sets the float value to the given parameter index.
public void setDouble(int paramIndex, double value)	sets the double value to the given parameter index.
public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

Example: of PreparedStatement interface that inserts the record

First of all create table as given below:

```
create table emp(id number(10),name varchar2(50));
```

Now insert records in this table by the code given below:

```
import java.sql.*;
class InsertPrepared
{
    public static void main(String args[])
    {
        try
        {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/Employee","root","admin@123");
            PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
            stmt.setInt(1,101);    //1 specifies the first parameter in the query
            stmt.setString(2,"Ratan");

            int i=stmt.executeUpdate();
            System.out.println(i+" records inserted");
            con.close();
        } catch (Exception e) { System.out.println(e);}
    }
}
```

Example of PreparedStatement interface that updates the record

```
PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?");
stmt.setString(1,"Sourav");    //1 specifies the first parameter in the query i.e. name
stmt.setInt(2,101);
int i=stmt.executeUpdate();
System.out.println(i+" records updated");
```

Example of PreparedStatement interface that deletes the record

```
PreparedStatement stmt=con.prepareStatement("delete from emp where id=?");
stmt.setInt(1,101);
int i=stmt.executeUpdate();
System.out.println(i+" records deleted");
```

Example of PreparedStatement interface that retrieve the records of a table

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

Example of PreparedStatement to insert records until user press n :

```
import java.sql.*;
import java.io.*;
class RS{
    public static void main(String args[])throws Exception{
        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/
                                                    Employee","root","admin@123");

        PreparedStatement ps=con.prepareStatement("insert into emp values(?,?,?)");
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

        do{
            System.out.println("enter id:");
            int id=Integer.parseInt(br.readLine());
            System.out.println("enter name:");
            String name=br.readLine();
            System.out.println("enter salary:");
            float salary=Float.parseFloat(br.readLine());

            ps.setInt(1,id);
            ps.setString(2,name);
            ps.setFloat(3,salary);
            int i=ps.executeUpdate();
            System.out.println(i+" records affected");

            System.out.println("Do you want to continue: y/n");
            String s=br.readLine();
            if(s.startsWith("n")){
                break;
            }
        }
```

```
}while(true);
```

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

```
}}
```

Java CallableStatement Interface:

The CallableStatement of JDBC API is used to call a stored procedure.

The prepareCall() method of connection interface will be used to create CallableStatement object.

Syntax:

```
public CallableStatement prepareCall("{ call procedurename(?,?,...?)}");
```

A Callable statement can have input parameters, output parameters or both. To pass input parameters to the procedure call you can use place holder and set values to these using the setter methods (setInt(), setString(), setFloat()) provided by the CallableStatement interface.

The registerOutParameter method of CallableStatement interface, that registers the output parameter with its corresponding type. It provides information to the CallableStatement about the type of result being displayed. The Types class defines many constants such as INTEGER, VARCHAR, FLOAT, DOUBLE, BLOB, CLOB etc.

Executing the Callable Statement

Once you have created the CallableStatement object you can execute it using one of the **execute()** method.

```
stmt.execute()
```

Example:

1) Create a table

```
create table emp2(id number(10), name varchar2(200));
```

2) Creating Procedure:

```
DELIMITER $$  
CREATE PROCEDURE `Employee`.`getEmpName` (IN EMP_ID INT, OUT EMP_FIRST  
    VARCHAR(50))  
BEGIN  
    SELECT first INTO EMP_FIRST FROM emp2 WHERE ID = EMP_ID;
```

```
END $$
```

```
DELIMITER ;
```

3) Write a java code to call the procedure 'getEmpName()'

```
import java.sql.*;

public class Proc {

    public static void main(String[] args) throws Exception{
        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/
            Employee","root","admin@123");

        CallableStatement stmt=con.prepareCall("{call getEmpName (?,?)}");
        stmt.setInt(1,1011);
        //Because second parameter is OUT so register it
        Stmt.registerOutParameter(2, java.sql.Types.VARCHAR);
        stmt.execute();
        String empName = stmt.getString(2);
        System.out.println("Emp Name = "+ empName);
        System.out.println("success");
    }
}
```

Now check the table in the database, value is inserted in the emp2 table.

Example to call the function using JDBC

In this example, we are calling the sum4 function that receives two input and returns the sum of the given number. Here, we have used the **registerOutParameter** method of CallableStatement interface, that registers the output parameter with its corresponding type. It provides information to the CallableStatement about the type of result being displayed.

The **Types** class defines many constants such as INTEGER, VARCHAR, FLOAT, DOUBLE, BLOB, CLOB etc.

Let's create the simple function in the database first.

```
create or replace function sum4 (n1 in number,n2 in number) return number is
temp number(8);
begin
    temp :=n1+n2;
```



```

        return temp;
    end;
/

```

Now, let's write the simple program to call the function.

```

import java.sql.*;
public class FuncSum {
    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");
        CallableStatement stmt=con.prepareCall("{?= call sum4(?,?)}");
        stmt.setInt(2,10);
        stmt.setInt(3,43);
        stmt.registerOutParameter(1,Types.INTEGER);
        stmt.execute();
        System.out.println(stmt.getInt(1));
    }
}

```

Output: 53

The ResultSet Interface

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. Initially, cursor points to before the first row.

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.

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

1. `createStatement(int RSType, int RSConcurrency);`
2. `prepareStatement(String SQL, int RSType, int RSConcurrency);`
3. `prepareCall(String sql, int RSType, int RSConcurrency);`

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.

Navigating a Result Set

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

S.N.	Methods & Description
1	public void beforeFirst() throws SQLException Moves the cursor just before the first row.
2	public void afterLast() throws SQLException Moves the cursor just after the last row.

3	public boolean first() throws SQLException Moves the cursor to the first row.
4	public void last() throws SQLException Moves the cursor to the last row.
5	public boolean absolute(int row) throws SQLException 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.

Getter Methods:

ResultSet has stored the data of the table from the Database. Getter methods are used to get the values of the table in ResultSet. For that, we need to pass either column Index value or Column Name.

The following are the getter methods in ResultSet:

- **int getInt(int ColumnIndex):** It is used to get the value of the specified column Index as an int data type.

<ul style="list-style-type: none"> • int getInt(String ColumnName): It is used to get the value of the specified column as an int data type.
<ul style="list-style-type: none"> • float getFloat(int ColumnIndex): It is used to get the value of the specified column Index as a float data type.
<ul style="list-style-type: none"> • float getFloat(String ColumnName): It is used to get the value of the specified column as a float data type
<ul style="list-style-type: none"> • String getString(int columnIndex): is used to return the data of specified column index of the current row as String.
<ul style="list-style-type: none"> • String getString(String columnName): is used to return the data of specified column name of the current row as String.
<ul style="list-style-type: none"> • java.sql.date getDate(int ColumnIndex): It is used to get the value of the specified column Index as a date value.
<ul style="list-style-type: none"> • Java.sql.date getDate(String ColumnName): It is used to get the value of the specified column as a date value.

There are getter methods for all primitive data types (Boolean, long, double) and String also in ResultSet interface.

Example:

```
import java.sql.*;
class FetchRecord
{
    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");
        Statement stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
        ResultSet rs=stmt.executeQuery("select * from emp765");

        //getting the record of 3rd row
        rs.absolute(3);
        System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));

        con.close();
    }
}
```


JDBC Updatable ResultSet:

Whenever we create a ResultSet object which never allows us to update the database through ResultSet object and it allows retrieving the data only in forward direction. Such type of ResultSet is known as non-updatable and non-scrollable ResultSet.

In order to make the ResultSet object as updatable and scrollable we must use the following constants which are present in ResultSet interface.

int Type	int Mode
TYPE_SCROLL_SENSITIVE	CONCUR_UPDATABLE

The above two constants must be specified while we are creating Statement object by using the following method:

`java.sql.Connection` 
`public Statement createStatement (int Type, int Mode);`

For example:

```
Statement          st=con.createStatement          (ResultSet.TYPE_SCROLL_SENSITIVE,  
ResultSet.CONCUR_UPDATABLE);
```

On ResultSet we can perform the following three operations, they are inserting a record, deleting a record and updating a record.

Steps for INSERTING a record through ResultSet object:

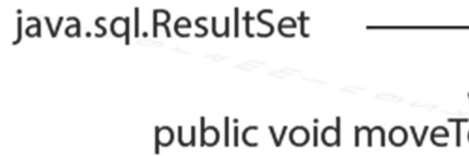
1. Decide at which position we are inserting a record by calling absolute method.

For example:

```
rs.absolute (3);
```

2. Since we are inserting a record we must use the following method to make the ResultSet object to hold the record.

java.sql.ResultSet



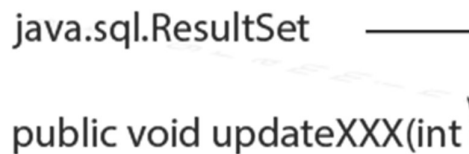
public void moveToInsertRow();

For example:

```
rs.moveToInsertRow ();
```

3. Update all columns of the database or provide the values to all columns of database by using the following generalized method which is present in ResultSet interface.

java.sql.ResultSet



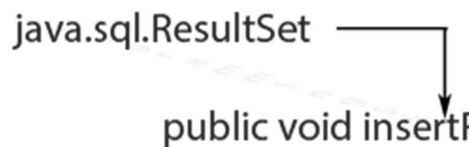
public void updateXXX(int colno, XXX value);

For example:

```
rs.updateInt (1, 5);  
rs.updateString (2, "abc");  
rs.updateInt (3, 80);
```

4. Upto step-3 the data is inserted in ResultSet object and whose data must be inserted in the database permanently by calling the following method:

java.sql.ResultSet



public void insertRow();

It throws an exception called SQLException.

For example:

```
rs.insertRow ();
```

Steps for DELETING a record through ResultSet object:

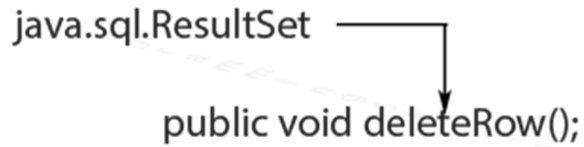
1. Decide which record you want to delete.

For example:

```
rs.absolute (3); // rs pointing to 3rd record & marked for deletion
```

2. To delete the record permanently from the database we must call the following method which is present in ResultSet interface.

java.sql.ResultSet



public void deleteRow();

For example:

```
rs.deleteRow ();
```

Steps for UPDATING a record through ResultSet object:

1. Decide which record to update.

For example:

```
rs.absolute (2);
```

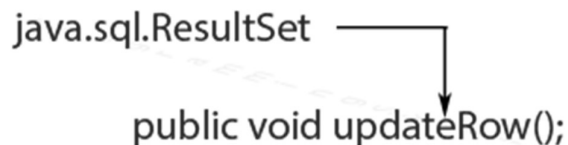
2. Decide which columns to be updated.

For example:

```
rs.updateString (2, "pqr");  
rs.updateInt (3, 91);
```

3. Using step-2 we can modify the content of ResultSet object and the content of ResultSet object must be updated to the database permanently by calling the following method which is present in ResultSet interface.

java.sql.ResultSet



public void updateRow();

For example:

```
rs.updateRow ();
```

For example:

Write a java program which illustrates the concept of updatable ResultSet?

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

```

class UpdateResultSet {

    public static void main(String[] args) {
        try {
            Class.forName("Sun.jdbc.odbc.JdbcOdbcDriver");
            System.out.println("DRIVERS LOADED...");
            Connection con = DriverManager.getConnection("jdbc:odbc:oradsn", "scott", "tiger");
            System.out.println("CONNECTION ESTABLISHED...");
            Statement st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_UPDATABLE);
            ResultSet rs = st.executeQuery("select * from emp1");
            rs.next();
            rs.updateInt(2, 8000);
            rs.updateRow();
            System.out.println("1 ROW UPDATED...");
            rs.moveToInsertRow();
            rs.updateInt(1, 104);
            rs.updateInt(2, 2000);
            rs.insertRow();
            System.out.println("1 ROW INSERTED...");
            rs.absolute(2);
            rs.deleteRow();
            System.out.println("1 ROW DELETED...");
            con.close();
        } catch (Exception e) {

            e.printStackTrace();
        }
    } // main
} // UpdateResultSet

```

Note: The scrollability and updatability of a ResultSet depends on the development of the driver of the driver vendors. OracleDriver and JdbcOdbcDriver will support the concept of scrollability and

updatability of a ResultSet but there may be some drivers which are available in the industry which are not supporting the concept of scrollability and updatability.

ResultSetMetaData Interface:

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

Commonly used methods of ResultSetMetaData interface

Method	Description
public int getColumnCount()throws SQLException	it returns the total number of columns in the ResultSet object.
public String getColumnName(int index)throws SQLException	it returns the column name of the specified column index.
public String getColumnType(int index)throws SQLException	it returns the column type name for the specified index.
public String getTableName(int index)throws SQLException	it returns the table name for the specified column index.

How to get the object of ResultSetMetaData:

The getMetaData() method of ResultSet interface returns the object of ResultSetMetaData.

Syntax:

public ResultSetMetaData getMetaData()**throws** SQLException

Example of ResultSetMetaData interface :

```
import java.sql.*;
class Rsmd
{
    public static void main(String args[])
    {
```

```

try
{
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection con=DriverManager.getConnection(
        "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

    PreparedStatement ps=con.prepareStatement("select * from emp");
    ResultSet rs=ps.executeQuery();
    ResultSetMetaData rsmd=rs.getMetaData();

    System.out.println("Total columns: "+rsmd.getColumnCount());
    System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));
    System.out.println("Column Type Name of 1st column: "+rsmd.getColumnTypeName(1));

    con.close();
} catch (Exception e) { System.out.println(e);}
}
}

```

```

Output: Total columns: 2
       Column Name of 1st column: ID
       Column Type Name of 1st column: NUMBER

```

Transaction Management in JDBC

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.

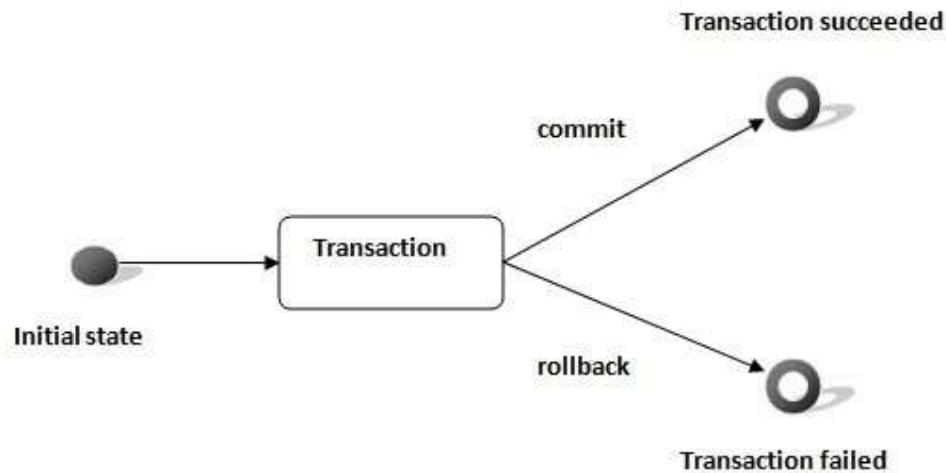
Consistency ensures bringing the database from one consistent state to another consistent state.

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.

Advantage of Transaction Management

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.

Simple 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 in jdbc using PreparedStatement

Let's see the simple 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:1521: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.

JDBC TYPES:

Java has a data type system, for example, int, long, float, double, string.

Database systems also have a type system, such as int, char, varchar, text, blob, clob.

The JDBC driver can convert the Java data type to the appropriate database type back and forth

Example:

MySQL Connector/J is flexible in the way it handles conversions between MySQL data types and Java data types.

In general, any MySQL data type can be converted to a `java.lang.String`, and any numeric type can be converted to any of the Java numeric types, although round-off, overflow, or loss of precision may occur.

Possible Conversions Between MySQL and Java Data Types

These MySQL Data Types	Can always be converted to these Java types
CHAR, VARCHAR, BLOB, TEXT, ENUM, and SET	<code>java.lang.String</code> , <code>java.io.InputStream</code> , <code>java.io.Reader</code> , <code>java.sql.Blob</code> , <code>java.sql.Clob</code>
FLOAT, REAL, DOUBLE PRECISION, NUMERIC, DECIMAL, TINYINT, SMALLINT, MEDIUMINT, INTEGER, BIGINT	<code>java.lang.String</code> , <code>java.lang.Short</code> , <code>java.lang.Integer</code> , <code>java.lang.Long</code> , <code>java.lang.Double</code> , <code>java.math.BigDecimal</code>
DATE, TIME, DATETIME, TIMESTAMP	<code>java.lang.String</code> , <code>java.sql.Date</code> , <code>java.sql.Timestamp</code>

The mapping is used when calling the `setXXX()` method from the `PreparedStatement` or `CallableStatement` object or the `ResultSet.updateXXX()/getXXX()` method.

SQL	JDBC/Java	setXXX	getXXX	updateXXX
VARCHAR	<code>java.lang.String</code>	<code>setString</code>	<code>getString</code>	<code>updateString</code>
CHAR	<code>java.lang.String</code>	<code>setString</code>	<code>getString</code>	<code>updateString</code>
LONGVARCHAR	<code>java.lang.String</code>	<code>setString</code>	<code>updateString</code>	
BIT	<code>boolean</code>	<code>setBoolean</code>	<code>getBoolean</code>	<code>updateBoolean</code>

NUMERIC	java.math.BigDecimal	setBigDecimal	getBigDecimal	updateBigDecimal
TINYINT	byte	setByte	getByte	updateByte
SMALLINT	short	setShort	getShort	updateShort
INTEGER	int	setInt	getInt	updateInt
BIGINT	long	setLong	getLong	updateLong
REAL	float	setFloat	getFloat	updateFloat
FLOAT	float	setFloat	getFloat	updateFloat
DOUBLE	double	setDouble	getDouble	updateDouble
VARBINARY	byte[]	setBytes	getBytes	updateBytes
BINARY	byte[]	setBytes	getBytes	updateBytes
DATE	java.sql.Date	setDate	getDate	updateDate
TIME	java.sql.Time	setTime	getTime	updateTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	getTimestamp	updateTimestamp
CLOB	java.sql.Clob	setClob	getClob	updateClob
BLOB	java.sql.Blob	setBlob	getBlob	updateBlob

Example

The following examples shows how to convert Java Date and Time classes to match the SQL data type.

```
public class Main {
```

```

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

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

    // SQL TIME
    java.sql.Time sqlTime = new java.sql.Time(javaTime);
    System.out.println("The SQL TIME is: " + sqlTime.toString());

    // SQL TIMESTAMP
    java.sql.Timestamp sqlTimestamp = new java.sql.Timestamp(javaTime);
    System.out.println("The SQL TIMESTAMP is: " + sqlTimestamp.toString());
}

```

The code above generates the following result.

```

The Java Date is:Sat Jun 27 18:33:51 PDT 2015
The SQL DATE is: 2015-06-27
The SQL TIME is: 18:33:51
The SQL TIMESTAMP is: 2015-06-27 18:33:51.366

```

Executing SQL Queries

The **Statement** interface provides methods to execute queries with the database.

Commonly used methods of Statement interface:

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.

```
import java.sql.*;
class ExQuery{
public static void main(String args[])throws Exception{
    Class.forName("com.mysql.cj.jdbc.Driver");
    Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/
                                                Employee", "root", "admin@123");

    Statement stmt=con.createStatement();
    int result=stmt.executeUpdate("delete from emp765 where id=33");
    System.out.println(result+" records affected");
    con.close();
}}
```

Executing SQL Updates

The **executeUpdate()** method returns the number of rows affected by the SQL statement (an INSERT typically affects one row, but an UPDATE or DELETE statement can affect more).

Example: of PreparedStatement interface that inserts the record

First of all create table as given below:

```
create table emp(id number(10),name varchar2(50));
```

Now insert records in this table by the code given below:

```
import java.sql.*;
class InsertPrepared
{
    public static void main(String args[])
    {
        try
        {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/
                                                        Employee", "root", "admin@123");

            PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
            stmt.setInt(1,101);    //1 specifies the first parameter in the query
        }
    }
}
```

```

        stmt.setString(2,"Ratan");

        int i=stmt.executeUpdate();
        System.out.println(i+" records inserted");
        con.close();
    } catch (Exception e) { System.out.println(e);}
    }
}

```

Example of PreparedStatement interface that updates the record

```

PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?");
stmt.setString(1,"Sourav"); //1 specifies the first parameter in the query i.e. name
stmt.setInt(2,101);
int i=stmt.executeUpdate();
System.out.println(i+" records updated");

```

Example of PreparedStatement interface that deletes the record

```

PreparedStatement stmt=con.prepareStatement("delete from emp where id=?");
stmt.setInt(1,101);
int i=stmt.executeUpdate();
System.out.println(i+" records deleted");

```

Example of PreparedStatement interface that retrieve the records of a table

```

PreparedStatement stmt=con.prepareStatement("select * from emp");
ResultSet rs=stmt.executeQuery();
while(rs.next()){
    System.out.println(rs.getInt(1)+" "+rs.getString(2));
}

```

java.sql package:

The java. sql package provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java programming language. This API

includes a framework whereby different drivers can be installed dynamically to access different data sources.

The java.sql package contains API for the following:

1. Making a connection with a database via the DriverManager facility

- [DriverManager](#) class -- makes a connection with a driver
- [SQLPermission](#) class -- provides permission when code running within a Security Manager, such as an applet, attempts to set up a logging stream through the DriverManager
- [Driver](#) interface -- provides the API for registering and connecting drivers based on JDBC technology ("JDBC drivers"); generally used only by the DriverManager class
- [DriverPropertyInfo](#) class -- provides properties for a JDBC driver; not used by the general user

2. Sending SQL statements to a database

- [Statement](#) -- used to send basic SQL statements
- [PreparedStatement](#) -- used to send prepared statements or basic SQL statements (derived from Statement)
- [CallableStatement](#) -- used to call database stored procedures (derived from PreparedStatement)
- [Connection](#) interface -- provides methods for creating statements and managing connections and their properties
- [Savepoint](#) -- provides savepoints in a transaction

3. Retrieving and updating the results of a query

- [ResultSet interface](#)

4. Standard mappings for SQL types to classes and interfaces in the Java programming language

- [Array](#) interface -- mapping for SQL ARRAY
- [Blob](#) interface -- mapping for SQL BLOB
- [Clob](#) interface -- mapping for SQL CLOB
- [Date](#) class -- mapping for SQL DATE
- [NClob](#) interface -- mapping for SQL NCLOB
- [Ref](#) interface -- mapping for SQL REF
- [RowId](#) interface -- mapping for SQL ROWID
- [Struct](#) interface -- mapping for SQL STRUCT
- [SQLXML](#) interface -- mapping for SQL XML
- [Time](#) class -- mapping for SQL TIME
- [Timestamp](#) class -- mapping for SQL TIMESTAMP

- Types class -- provides constants for SQL types

5. Custom mapping an SQL user-defined type (UDT) to a class in the Java programming language

1. SQLData interface -- specifies the mapping of a UDT to an instance of this class
2. SQLInput interface -- provides methods for reading UDT attributes from a stream
3. SQLOutput interface -- provides methods for writing UDT attributes back to a stream

6. Metadata

- [DatabaseMetaData](#) interface -- provides information about the database
- [ResultSetMetaData](#) interface -- provides information about the columns of a ResultSet object
- ParameterMetaData interface -- provides information about the parameters to PreparedStatement commands.

7. Exceptions

- [SQLException](#) -- thrown by most methods when there is a problem accessing data and by some methods for other reasons
- [SQLWarning](#) -- thrown to indicate a warning
- [DataTruncation](#) -- thrown to indicate that data may have been truncated
- [BatchUpdateException](#) -- thrown to indicate that not all commands in a batch update executed successfully

What is the difference between stored procedures and functions:

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.

MySQL Commands:

What is SQL?

SQL is the standard language for dealing with Relational Databases.

MySQL Create Database

MySQL create database is used to create database. For example

```
create database db1;
```

MySQL Select/Use Database

MySQL use database is used to select database. For example

```
use db1;
```

MySQL Create Query

MySQL create query is used to create a table, view, procedure and function. For example:

```
CREATE TABLE customers  
(id int(10),  
  name varchar(50),  
  city varchar(50),  
  PRIMARY KEY (id )  
);
```

MySQL Alter Query

MySQL alter query is used to add, modify, delete or drop columns of a table. Let's see a query to add column in customers table:

```
ALTER TABLE customers  
ADD age varchar(50);
```

MySQL Truncate Table Query

MySQL update query is used to truncate or remove records of a table. It doesn't remove structure. For example:

```
truncate table customers;
```

MySQL Drop Query

MySQL drop query is used to drop a table, view or database. It removes structure and data of a table if you drop table. For example:

```
drop table customers;
```

MySQL Insert Query

MySQL insert query is used to insert records into table. For example:

```
insert into customers values(101,'rahul','delhi');
```

MySQL Update Query

MySQL update query is used to update records of a table. For example:

```
update customers set name='bob', city='london' where id=101;
```

MySQL Delete Query

MySQL update query is used to delete records of a table from database. For example:

```
delete from customers where id=101;
```

MySQL Select Query

Oracle select query is used to fetch records from database. For example:

```
SELECT * from customers;
```