JDBC :

1. Introduction
    - DriverManager
    - Driver types
2. Getting Started with JDBC
    - Setting up Development Environment
    - Connection
    - Managing Database Resources
    - Handling JDBC Exceptions
3. CRUD operations using JDBC
    - Statement
    - ResultSet
    - PreparedStatement
4. Working with Stored Procedures
    - Callable statement
    - IN & OUT parameters
5. Managing Transactions
6. Working with BLOB and CLOB
    - Store and retrieve images, files from Database
7. Working with Metadata
8. Pooling Database connections

JDBC is a API for connecting programs written in Java to any database.

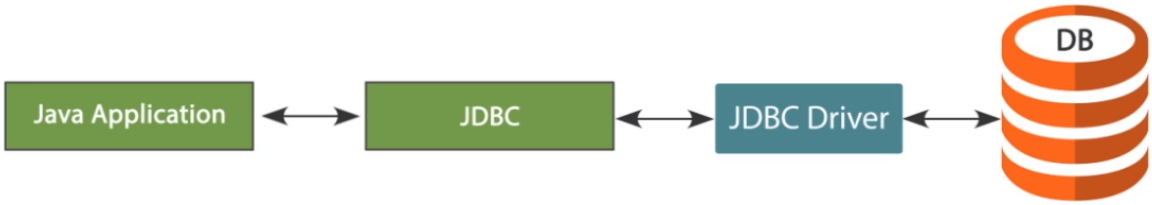
4 components that make up JDBC Architecture :

- JDBC Client – the app that we are developing
- JDBC API
- JDBC Driver – interface between our code and the underlying database.
- JDBC DriverManager – helper class which finds the driver and establishes a connection to the database.

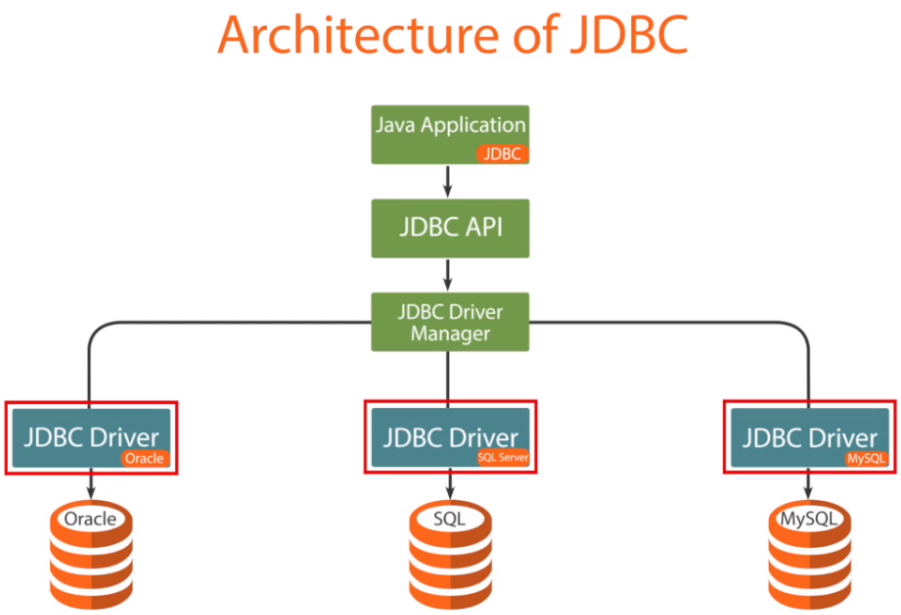
JDBC API is part of **java.sql.*, and javax.sql.*** packages. The API comprises of many interfaces and classes(like, Connection, Statement, ResultSet, etc) that we use to interact with Database.

JDBC is oriented towards Relational Datatabases, and it **uses JDBC Drivers** to connect with the database.

JDBC Drivers : These drivers implement the JDBC specification and is the interface between client code and the DB.



Architecture of JDBC :



DriverManager – it is the basic service for managing a set of JDBC drivers. It is used to match the connection request from a Java application with the proper database driver using communication sub-protocol.

Our JDBC Client code uses the DriverManager helper class ONLY ONCE to find the appropriate Driver class to connect to the database and it establishes a connection through the driver and returns the connection back to the Client code.

Whenever we want to connect to the database from Java applicaiton, the very first step is to load the relevant JDBC Driver to the Driver Manager.

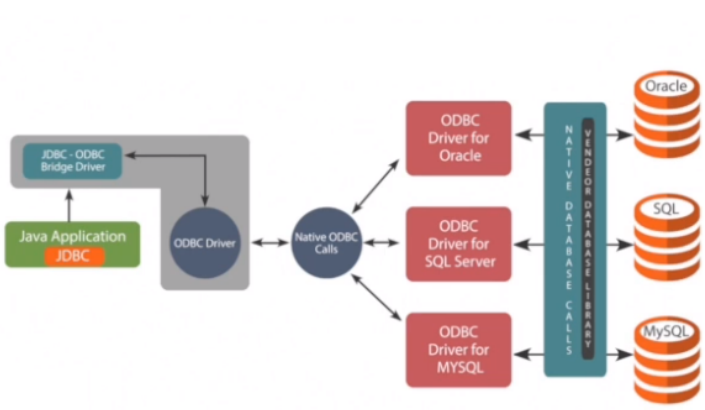
How do you load?? Using : Class.forName("com.mysql.jdbc.Driver"); Class.forName() method is used to load and register a driver with the DriverManager.

From JDBC 4.0 onwards, applications no longer need to explicitly load JDBC Drivers. getConnection() method of DriverManager will locate suitable driver.

Types of JDBC Drivers :

1. **Type 1: JDBC-ODBC bridge driver**
2. **Type 2: Native-API driver**
3. **Type 3: Network-protocol driver (Middleware driver)**
4. **Type 4: Database-protocol driver (Pure Java driver) – widely used**

Type 1 : JDBC – ODBC bridge driver



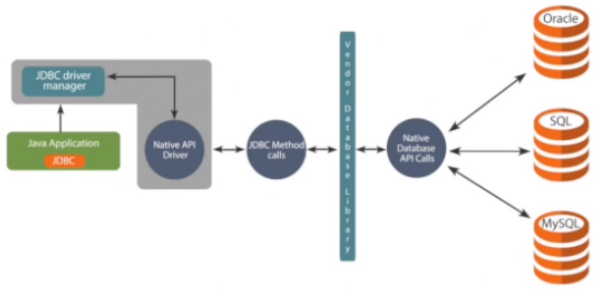
Advantages

- It is very easy to use
- Almost any database is supported

Limitations

- Performance will not be efficient
- ODBC Driver needs to be installed
- Type 1 drivers are not portable

Type 2 : Native – API driver



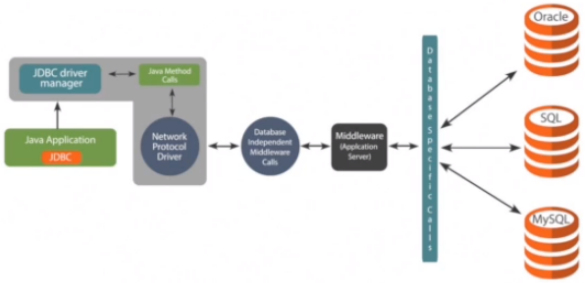
Advantages

- Faster than Type 1 Driver

Limitations

- Client Side Library is not available for all databases
- Vendor Client Library needs to be installed
- It is a Platform Dependent

Type 3 : Network-protocol Driver (Middleware driver)



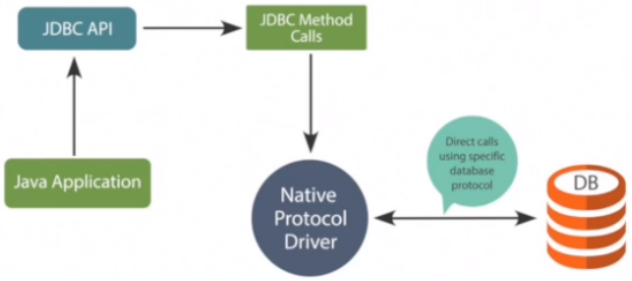
Advantages

- No additional library installation is required on client system
- No changes are required at client for any DB
- Supports Caching of Connection, Query Results, Load Balancing, Logging and Auditing etc.
- A Single Driver can handle any database provided the middleware supports it

Limitations

- Performance will be slow
- Requires Database-specific coding
- Maintenance of Network Protocol driver becomes costly

Type 4 : Database-protocol driver (Pure Java driver)



Advantages

- Platform Independent
- No intermediate format is required
- Application connects directly to the database server
- Performance will be very fast
- JVM manage all aspects

Limitations

- Drivers are database dependent

Usage of Database drivers :

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

Oracle commands : If you have installed **Oracle Express 11g**. Running some commands :

Go to : 'Run SQL Command Line'

```
SQL> connect system/oracle
Connected.
SQL>
```

In the above, 'oracle' is the password given during installation to the account 'system'.

Default users provided by the Oracle Database.

```
SQL>select username, password from dba_users;
```

Alter user 'hr' password :

```
SQL>alter user hr identified by hr;
User altered.
SQL>
```

Connect using hr user :

```
SQL>connect hr/hr
Connected
SQL>
```

Verify existing tables within the user hr :

```
SQL>select table_name from user_tables;
SQL>select tname from tab;
```

How to connect to the Database :

To connect to the Oracle database, you need the Java driver. You can download the JDBC driver for Oracle database at this location :

http://www.oracle.com/technetwork/apps-tech/jdbc-112010-090769.html

Download the driver JAR file and **add it to the build path of your Project**. Now it is available for use within your application.

Manage Database Resources :

This is to efficiently manage the database resources within your Java application.

DBType.java :

```
public enum DBType {
	ORADB, MYSQLDB
}
```

DBUtil.java :

```
public class DBUtil {
	private static final String oraUser="hr";
	private static final String oraPwd ="hr";
	private static final String mySqlUser = "root";
	private static final String mySqlPwd = "root";
	private static final String oraCS = "jdbc:oracle:thin:@localhost:1521:xe";
	private static final String mySQLCS = "jdbc:mysql://localhost:3306/world";

	public static Connection getConnection(DBType dbType) throws SQLException {
		switch (dbType) {
		    case ORADB:
			return DriverManager.getConnection(oraCS, oraUser, oraPwd);
		    case MYSQLDB:
			return DriverManager.getConnection(mySQLCS, mySqlUser, mySqlPwd);
		    default:
			return null;
		}
	}

	public static void showErrorMessage(SQLException e){
		System.err.println("Error :" + e.getMessage());
		System.err.println("Error Code :" + e.getErrorCode());
	}
}
```

Test code :

TestManageDBResources.java :

```
public class TestManageDBResources {

	public static void main(String[] args) throws SQLException {

		Connection conn = null;
		try {
			//conn = DriverManager.getConnection(dbUrl, username, password);
			conn = DBUtil.getConnection(DBType.MYSQLDB);

			System.out.println("Connection Established to MYSQL Database");

		} catch (SQLException e) {

			System.err.println(e.getMessage());
		}
		finally{
			conn.close();
		}
	}
}
```

JDBC Client code and the type of operations we do :

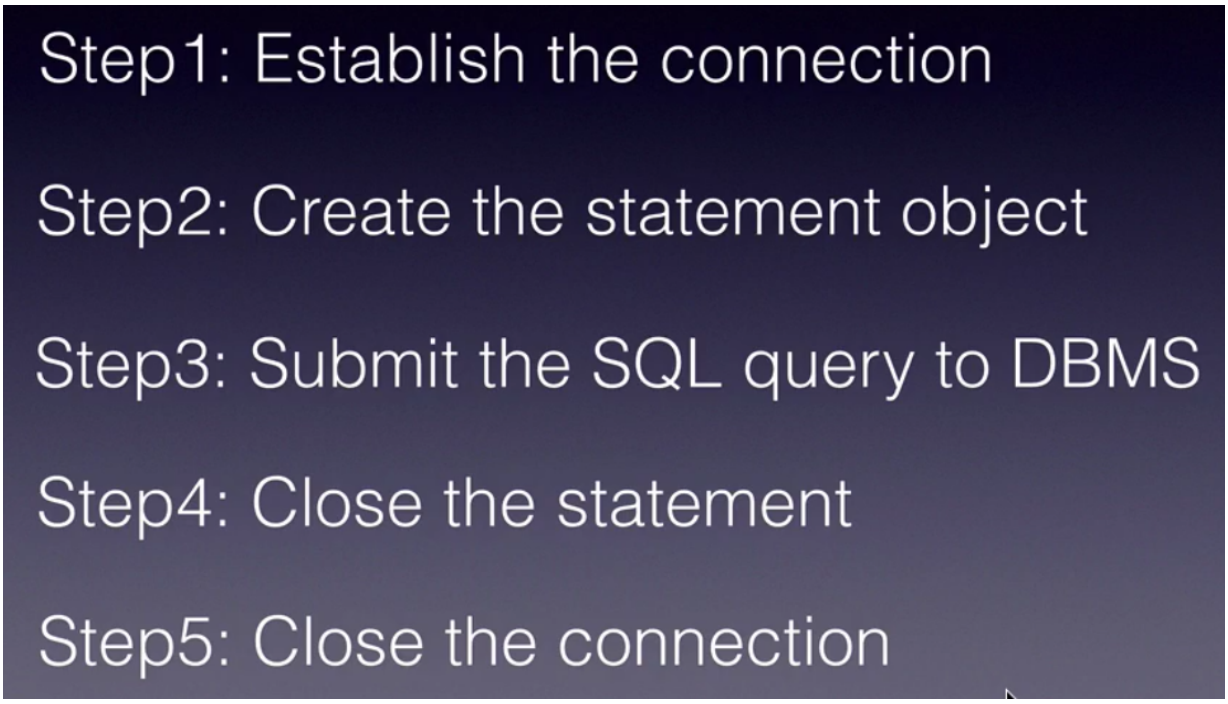
Create the Account table :

use mylocaldb;

create table account(accno int,lastname varchar(25),firstname varchar(25),bal int)

select * from account;

Steps to perform CRUD operations :



Create a simple Java project with name : JDBCBasics.
In this project, create a new folder named 'lib'.
Download the driver jar file : mysql-connector-java-5.1.41-bin.jar and add it to the project lib folder.

Add the driver JAR file to the build path -> Libraries. Then, the project will show the driver JAR file in Referenced Libraries.

Class.forName( ) is not needed since JDBC 4.0.

Any JDBC 4.0 drivers that are found in your class path are automatically loaded. (However, you must manually load any drivers prior to JDBC 4.0 with the method Class.forName.) In previous versions of JDBC, to obtain a connection, you first had to initialize your JDBC driver by calling the method `Class.forName`. This methods required an object of type `java.sql.Driver`. Each JDBC driver contains one or more classes that implements the interface `java.sql.Driver`.

Step 1: Connect to the Database

```
Connection myConn = DriverManager.getConnection(
                        "jdbc:mysql://localhost/mylocaldb", "root", "chotu123");
```

What is connection string ??

Connection String varies from database to database and is used to connect to the database.

For mySQL database : "jdbc:mysql://localhost/test"

For oracle database :

Using host:port:sid syntax

```
"jdbc:oracle:thin:@prodHost:1521:ORCL"
```

Using TNS syntax :

```
String connString = "jdbc:oracle:thin:@(description=(address_list=
  (address=(protocol=tcp)(port=1521)(host=prodHost)))
(connect_data=(INSTANCE_NAME=ORCL)))";
```

Step 2: Create Statement object

Statement is a key interface to create the DML and DQL statements. Used for executing static SQL queries.

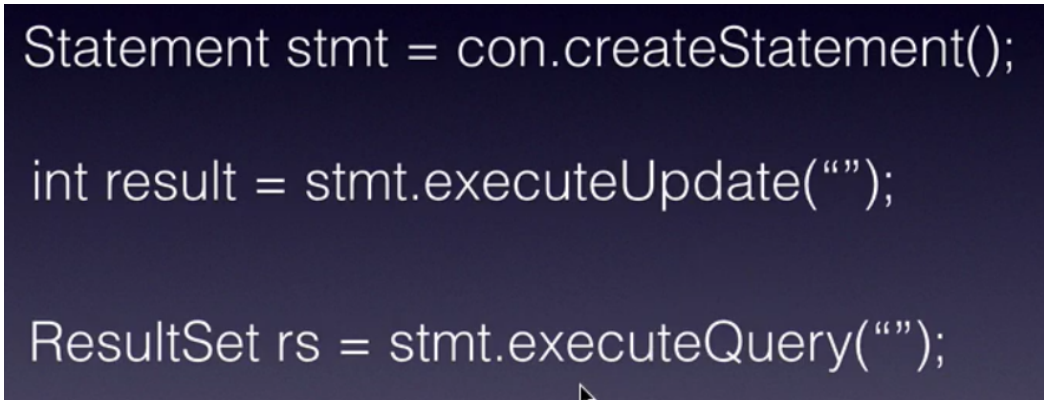
How do you create Statement object ??

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

Then, we can use this statement instance to execute SQL queries. 2 methods of importance.

For Insert, Update, Delete operations, --> use stmt.executeUpdate(" ") method. It returns an integer which indicate the number of records effected by the statement.

For select queries -> use stmt.executeQuery(" ") method. It returns a ResultSet object which contans the list of rows/records returned by the select query.



Insert a record into Account table in database :

We are going to Statement object and the method executeUpdate.

```
try {
	Connection myConn = DriverManager.getConnection(
		"jdbc:mysql://localhost/mylocaldb", "root", "chotu123");

	Statement statement = myConn.createStatement();
	int result = statement.executeUpdate(
		"insert into account values(1, 'Dayala', 'Raghu', 10000)");

	System.out.println(result + " rows got inserted..");

	myConn.close();
} catch (SQLException e) {
	e.printStackTrace();
}
```

Update a record :

```
int result = statement.executeUpdate(
		"update account set bal = 20000 where accno = 1");
System.out.println(result + " rows got updated..");
```

Delete a record :

```
int result = statement.executeUpdate("delete from account where accno = 1");

System.out.println(result + " rows got deleted..");
```

Executing Select Query :

```
public class TestStaticSQLStatement {

	public static void main(String[] args) throws SQLException {

		Connection conn = null;
		Statement stmt = null;
		ResultSet rs = null;

		try{
			conn = DBUtil.getConnection(DBType.MYSQLDB);
			stmt = conn.createStatement();
			rs = stmt.executeQuery("select * from Countries");
			rs.last(); // only works with MySQL. Check below explanation.
			System.out.println("Total Rows : " + rs.getRow());
		}catch(SQLException ex){
			DBUtil.showErrorMessage(ex);
		}
		finally{
			if( rs!= null )
				rs.close();
			if (stmt != null )
				stmt.close();
			if(conn != null)
				conn.close();
		}
	}
}
```

The nature of the SQL command is determined by the database you're working with.

Example : The default ResultSet type behavior will differ from one database to another. With MySQL, the default ResultSet object is scrollable, which means you can move the cursor up and down, going to the first row, the last row, and moving around as you like.

However with Oracle database, the default ResultSet type is forward only, meaning that the cursor will start before the data and you will be able to move forward to the end, but you won't be able to move back again.

From JDBC 4.0 onwards, we have this called **try-with-resources**.

ResultSets :

A ResultSet object maintains a cursor that points to the row in the ResultSet which helps navigating from one row to the other row present at the ResultSet.

When the ResultSet is first returned, the starting cursor position is before the first row of the data. That is, it is not pointing to any data at first. You have to explicitly move the cursor to the data in order to read it.

Navigational methods :

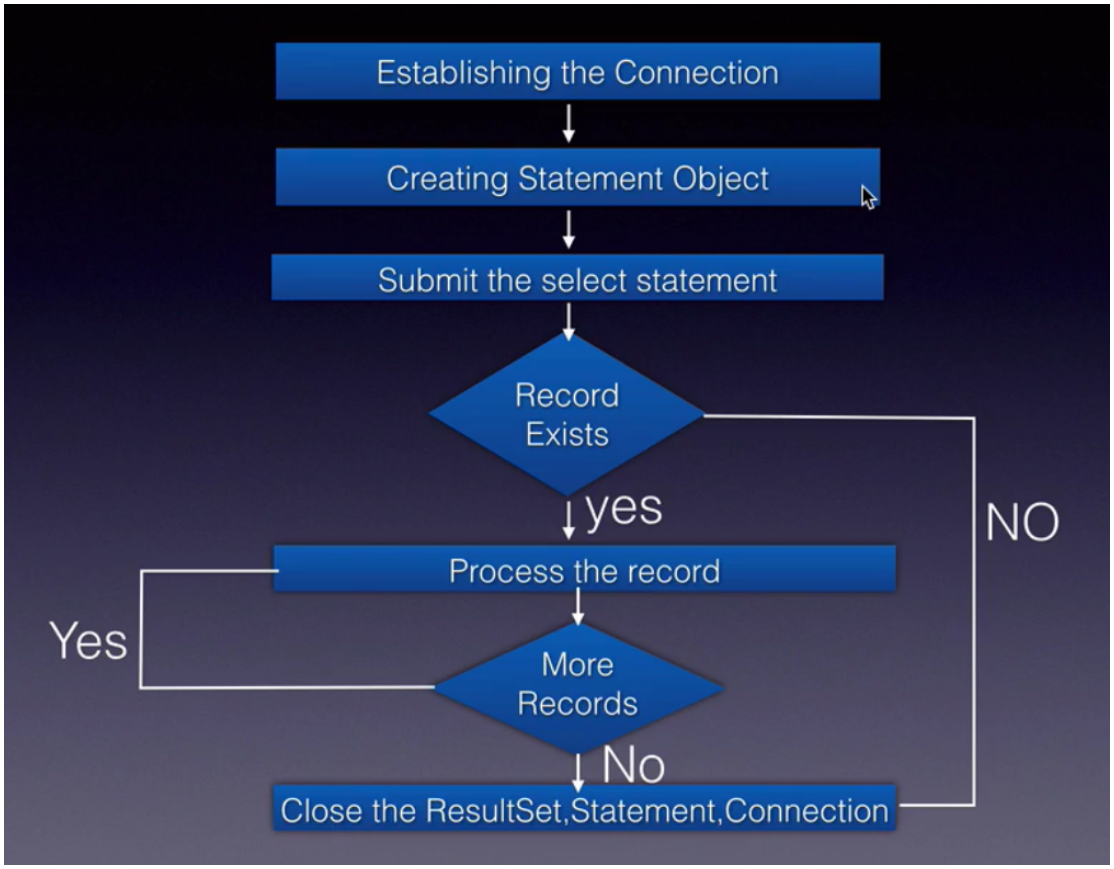
- beforeFirst(), afterLast()
- first(), last()
- previous(), next()
- getRow()

Get methods for getting column data :

- getXXX() method for each data type and each method has two versions : one that takes, column name and the other that takes a column index.

```
public class IteratingWithResultSetsUsingTryWithResource {
	public static void main(String[] args) throws SQLException {
		try(
			Connection conn = DBUtil.getConnection(DBType.ORADB);
			Statement stmt = conn.createStatement();
			ResultSet rs = stmt.executeQuery("Select * from Employees");
		) {
			String format = "%-4s%-20s%-25s%-10f\n";
			while (rs.next()) {
				System.out.format(format,
					rs.getString("Employee_ID"),
					rs.getString("First_Name"),
					rs.getString("Last_Name"),
					rs.getFloat("Salary"));
			}
		} catch (SQLException e) {
			DBUtil.showErrorMessage(e);
		}
	}
}
```

Reading data from DB using JDBC : Select Query :



When we execute the query, the returned rows will be a list of records stored in ResultSet object.

```
ResultSet rs = statement.executeQuery("select * from account");
while(rs.next()){
	// column index starts from 1
	System.out.print(rs.getString(2)+ " "); // lastname
	System.out.print(rs.getString(3)+ " "); // firstname
	System.out.println(rs.getString(4)); // balance
}
```

Understanding Scrollable ResultSet :

Helps to navigate efficiently within the ResultSet. ResultSet should be scrollable if we want to move in forward and backward directions.

The cursor is movable based on the properties of the ResultSet.

- TYPE_FORWARD_ONLY – default. Cursor can only move in the forward direction.
- TYPE_SCROLL_INSENSITIVE – if used, cursor can scroll forward and backward and the ResultSet is not sensitive to changes made by others to the database that occur after the ResultSet was created.
- TYPE_SCROLL_SENSITIVE – if used, cursor can scroll forward and backward and the ResultSet is sensitive to changes made by others to the database that occur after the ResultSet was created.

Whenever we have a requirement of performing navigations efficiently within the ResultSet, then it is mandatory to use ScrollableResultSet.

```
public class ResultSetScrollingDemo {
   public static void main(String[] args) throws SQLException  {
        try(     Connection conn = DBUtil.getConnection(DBType.ORADB);
                 Statement stmt = conn.createStatement(
                                      ResultSet.TYPE_SCROLL_INSENSITIVE,
                                      ResultSet.CONCUR_READ_ONLY);
                 ResultSet rs = stmt.executeQuery("Select * From Employees Where Rownum <= 10");
        )
        {
                 String format = "%-4s%-20s\n";
                 rs.beforeFirst();
                 System.out.println("First 10 Rows : ");
                 while(rs.next()){
                    System.out.format(format, rs.getString("Employee_ID"),rs.getString("First_Name"));
                 }
                 rs.afterLast();
                 System.out.println("Last 10 Rows : ");
                 while(rs.previous()){
                    System.out.format(format, rs.getString("Employee_ID"),rs.getString("First_Name"));
                 }

                 rs.first();
                 System.out.println("First Record :");
                 System.out.format(format, rs.getString("Employee_ID"),rs.getString("First_Name"));
                 rs.last();
                 System.out.println("Last Record :");
                 System.out.format(format, rs.getString("Employee_ID"),rs.getString("First_Name"));

                 rs.absolute(4);
                 System.out.println("Record at 4th Row :");
                 System.out.format(format, rs.getString("Employee_ID"),rs.getString("First_Name"));

                 rs.relative(2);
                 System.out.println("Record at 6th Row : ");
                 System.out.format(format, rs.getString("Employee_ID"),rs.getString("First_Name"));

                 rs.relative(-4);
                 System.out.println("Record at 2nd Row : ");
                 System.out.format(format, rs.getString("Employee_ID"),rs.getString("First_Name"));
        }
        catch(SQLException e){  DBUtil.showErrorMessage(e); }  }         }
```

Understanding Updatable ResultSets :

Updatable ResultSet allows **modification to data in a table through the ResultSet.**

For this, we have : updateXXX methods. Each method has two versions. One that takes, column name and the other one takes column index. All the updateXXX methods throws SQLException.

To update our changes to the row in the database, we need to invoke one of the following methods :

- updateRow() - updates the current row by updating the corresponding row in the database.
- deleteRow() - removes the current row from the database.
- refreshRow() - refreshes data in the ResultSet to reflect any recent changes in the database.
- cancelRowUpdates() - cancels any updates made to the current row.
- insertRow() - inserts a row in the database.

```
public class UpdatableResultSetDemo {
        public static void main(String[] args) throws SQLException{
          try(   Connection conn = DBUtil.getConnection(DBType.ORADB);
                Statement stmt = conn.createStatement(
                                        ResultSet.TYPE_SCROLL_INSENSITIVE,
                                        ResultSet.CONCUR_UPDATABLE);
                ResultSet rs = stmt.executeQuery("Select Department_Id, Department_Name,
Manager_Id, Location_Id from Departments");
                )
        {
                rs.absolute(6);

                rs.updateString("Department_Name", "Information Technology");
                rs.updateRow();
                System.out.println("Record Updated Successfully");

                rs.moveToInsertRow();
                rs.updateInt("Department_Id", 999);
                rs.updateString("Department_Name", "Training");
                rs.updateInt("Manager_Id", 200);
                rs.updateInt("Location_Id", 1800);
                rs.insertRow();
                System.out.println("Record Inserted Successfully");
        }
        catch(SQLException ex){
                DBUtil.showErrorMessage(ex);
        }       }       }
```

Clean up JDBC Resources :

Before Java 7, we need to explicitly close the resources in the finally block.

From Java 7 onwards, we can use try-with-resources block as the JDBC interfaces implement AutoCloseable.

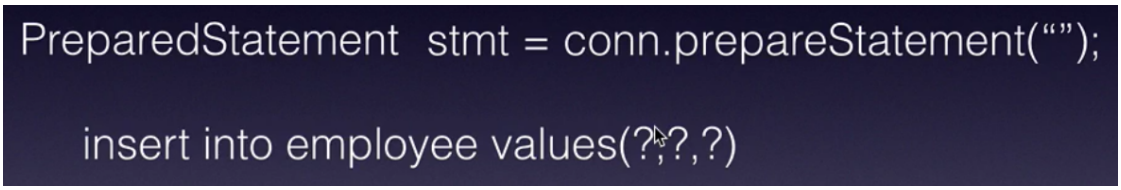
```
try(Connection myConn =
                  DriverManager.getConnection(
                        "jdbc:mysql://localhost/mylocaldb", "root", "chotu123");
                  Statement statement = myConn.createStatement();
                  ResultSet rs = statement.executeQuery("select * from account");
      ) {

            while(rs.next()){
                  // column index starts from 1
                  System.out.print(rs.getString(2)+ " "); // lastname
                  System.out.print(rs.getString(3)+ " "); // firstname
                  System.out.println(rs.getInt(4)); // balance
            }

      } catch (SQLException e) {
                  e.printStackTrace();
      }
```

JDBC PreparedStatement :

It is a pre-compiled version of a SQL Statement. Instead of hard-coding values in SQL query, we use ? symbols. These are nothing but placeholders.



Later to assign values for the placeholders, we use stmt.setXXX( ) methods. This process is called binding the values to the placeholders.

Compilation of the sql statement happens right at the call conn.prepareStatement(" "); This will improve the performance.

```
public void init(){
	try {
		Class.forName("com.mysql.jdbc.Driver");
		conn = DriverManager.getConnection(
				"jdbc:mysql://localhost/mylocaldb", "root", "chotu123");
		// this is a prepared statement, compiled only once
		stmt = conn.prepareStatement("insert into product values(?, ?, ?, ?)");
	} catch (ClassNotFoundException e) {
		e.printStackTrace();
	} catch (SQLException e) {
		e.printStackTrace();
	}
}
```

```
protected void doPost(HttpServletRequest request,
		HttpServletResponse response) throws ServletException, IOException {

	int id = Integer.parseInt(request.getParameter("id"));
	String name = request.getParameter("name");
	String desc = request.getParameter("description");
	int price = Integer.parseInt(request.getParameter("price"));

	try {
		stmt.setInt(1, id);
		stmt.setString(2,  name);
		stmt.setString(3,  desc);
		stmt.setInt(4,  price);

		int result = stmt.executeUpdate();
		if(result > 0){
			PrintWriter out = response.getWriter();
			response.setContentType("text/html");
			out.print("<b>Product created..</b>");
		}

	} catch (SQLException e) {
		e.printStackTrace();
	}
}
```

Prepared Statements :

- **Since SQL statements are pre-compiled, they improve performance of an Application**
- **Easy to set SQL Parameter value**
- **Prevents SQL Dependency Injection Attacks**

The advantage of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it.

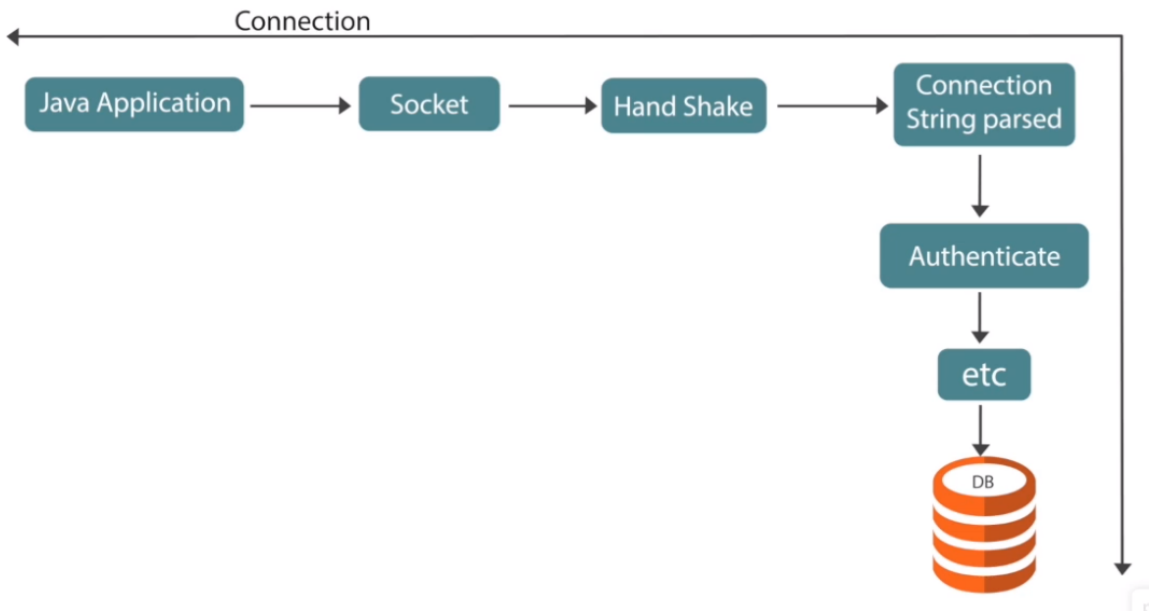
Instead of hard-coding parameter values in SQL query, we use ? symbols. These are nothing but placeholders.

The parameter positions are 1-based from L-to-R.

Understanding Connection Pooling :

We shall understand the importance of connection pooling and we shall understand how connection pooling works with respect to JDBC.

Whenever a Java application interacts with a database, the connection process consists of several time consuming steps.



- In simple terms to understand, a physical channel such as a socket or a named pipe, must be established.
- The initial handshake with a server must occur.
- The connection string information must be parsed.
- The connection must be authenticated by the server.
- Checks must be run for enlisting in the current transaction and so on.

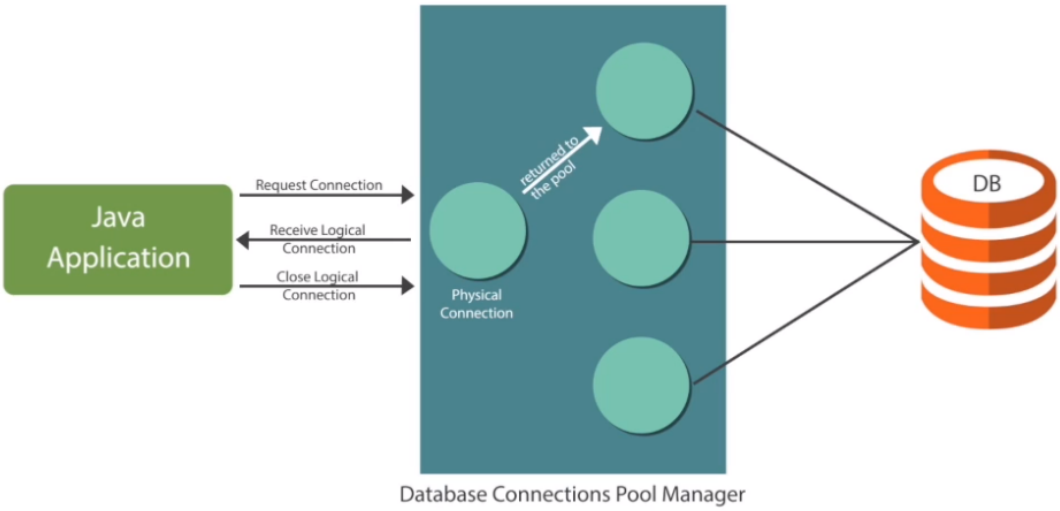
During application execution, many identical connections will be repeatedly opened and closed.

To **minimize the cost of opening connections**, JDBC uses an **optimization technique** called as **connection pooling**.

JDBC Connection Pool is a **factory that consists of readily available JDBC Connection objects** before actually being used. The advantage with JDBC Connection Pool is we can use minimum number of connection objects to give services to maximum clients.

Now let us understand how JDBC connection pooling works.

A **Pool Manager** creates the **initial physical connections** for the database.



When an application requires a connection, one of these physical connections is provided as a logical connection to the application. Application receives a logical connection. When the application is finished, the logical connection is disconnected, but the physical connection is returned to the pool for reuse such that the next application can use.

Pool Manager manages the distribution of the physical connections to the applications in the form of logical connections, returns any closed logical connections to the pool and handles any errors.

In earlier versions of JDBC, if there was a network error, then the physical connect had disconnected required a restart of the server. JDBC 4 corrected this pooling problem, it provides a way for the Pool Manager to ask a connection if it is still valid for this to work properly or not.

There are **two types** of JDBC Connection Pools :

- JDBC Driver Managed or Standalone JDBC Connection.
  - Oracle thin driver or OCI driver, gives one built in JDBC Connection Pool for Oracle.
- Web Server or Application Server Managed JDBC Connection.

While developing a small scale application like standalone Desktop applications we can use Driver Managed JDBC Connection Pool. While developing large scale applications like websites and enterprise applications, we can use Server Managed JDBC Connection Pool.

All JDBC Connection objects in a JDBC Connection Pool represents connectivity to the same database. That is, JDBC Connection Pool for Oracle means all JDBC Connection objects in the connection pool represents connectivity with Oracle database.

JDBC Datasource object means, it is the object of a Java class that implements **javax.sql.DataSource** interface. **JDBC Datasource object represents JDBC Connection Pool**. So our application should always get JDBC Connection object from the connection pool through Datasource object.

Now let us understand how to **implement connection pooling**.

```
public class ConnectionPoolingDemo {
        public static void main(String[] args) throws SQLException {
                OracleConnectionPoolDataSource ds = new OracleConnectionPoolDataSource();
                ds.setDriverType("thin");
                ds.setServerName("localhost");
                ds.setPortNumber(1521);
                ds.setServiceName("xe");
                ds.setUser("hr");
                ds.setPassword("hr");


                PooledConnection pconn = ds.getPooledConnection();
                Connection conn = pconn.getConnection();
                Statement stmt = conn.createStatement();
                ResultSet rs = stmt.executeQuery("Select * From Departments");


                String format = "%-30s%-50s%-25s\n";
                System.out.format(format,"Department #","Department Name","Location");
                System.out.format(format,"-------------","-----------------","-------------");


                while(rs.next()){
                        System.out.format(format,rs.getString("Department_ID"),
                                rs.getString("Department_Name"), rs.getString("Location_Id"));
                }


                rs.close();
                stmt.close();
                conn.close();
                pconn.close();
        }
}
```

1. The first step we need to perform is to **create an object for the datasource :**

   OracleConnectionPoolDataSource ds = new OracleConnectionPoolDataSource();

   A Datasource object is a factory for Connection objects.

   ** An object that implements the datasource interface will typically be registered with the **JNDI service provider**.

2. Set the values for the datasource object :

   ```
   ds.setDriverType("thin");
   ds.setServerName("localhost");
   ds.setPortNumber(1521);
   ds.setServiceName("xe");
   ds.setUser("hr");
   ds.setPassword("hr");
   ```

   It is always advisable to maintain the above details in a utility file or the property file instead of writing within the application. To make it simple for the demonstrating, I have mentioned these details within the program itself.

3. Then let us create a Pooled Connection object:

   PooledConnection pconn = ds.getPooledConnection();

   This will attempt to establish a pooled connection with the database.

There are many other ways to achieve connection pooling in JDBC by configuring the application server.

Server Managed JDBC Connection :

Connection pooling allows us to request the Container to create a set of JDBC connections right when it starts up. When container is starting up, we request it to create a set of JDBC connections.

These connections can then be used by Servlets or JSPs to interact with database and send the connection back when they are done with their work.

This connection pooling greatly enhances performance and also provides reuse of connections. (how reuse?? when servlet is done with its work, it will return the connection back to the pool).

Configuring Connection pooling :

1. step 1 : Copy Driver jar file to the Apache Tomcat installation lib folder. Tomcat needs this driver to establish connections.
2. Step 2 : Configure Resource element in context.xml file.

Context.xml file is under <ApacheTomcatInstallationDir>/conf folder.

C:\Program Files\Apache Software Foundation\Tomcat 8.0\conf

In the context.xml file, add the Resource element like this :

```
<Resource name="myds" auth="Container" type="javax.sql.DataSource"
    driverClassName="com.mysql.jdbc.Driver" url="jdbc:mysql://localhost/mydb" username="root" password="test"/>
```

Acquire Database connection from Connection pool :

1. Connect to the naming service
2. Look up for the DataSource
3. Get the connection

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException {
      try {
            // step 1: connect to the naming service
            Context namingContext = new InitialContext();
            // step 2: J2EE standard that lookup name should follow this prefix :
            // "java:comp/env/"
            // It is mandatory to follow the naming convention
            DataSource ds = (DataSource)namingContext.lookup("java:comp/env/myds");
            // step 3: Get connection from DataSource
            Connection conn = ds.getConnection();
            System.out.println(conn);
      } catch (NamingException e) {
            e.printStackTrace();
      } catch (SQLException e) {
            e.printStackTrace();
      }
      response.getWriter().append("Served at: ").append(request.getContextPath());
}
```