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 following database related tasks

* Making a connection to a database.
* Creating SQL or MySQL statements.
* Executing SQL or MySQL queries in the database.
* 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.

JDBC Architecture consists of two layers:

* JDBC API: This provides the application-to-JDBC Manager connection.
  + The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.
* JDBC Driver API: This supports the JDBC Manager-to-Driver Connection.
  + The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

The JDBC API provides the following interfaces and classes:

* DriverManager: This **class** manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication subprotocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
* Driver: This **interface** handles the communications with the database server. Instead of interacting directly with Driver objects, you will use DriverManager objects, which manages objects of this type.
* Connection: This **interface** with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
* Statement: You use objects created from this **interface** to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
* ResultSet: These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
* SQLException: This class handles any errors that occur in a database application.

The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. It offers the main classes for interacting with your data sources.

The new features in these packages include changes in the following areas:

* Automatic database driver loading.
* Exception handling improvements.
* Enhanced BLOB/CLOB functionality.
* Connection and statement interface enhancements.
* National character set support.
* SQL ROWID access.
* SQL 2003 XML data type support.
* Annotations.

To start developing with JDBC, you should setup your JDBC environment by following the steps shown below.

* Install Java
  + Set JAVA\_HOME to C:\Program Files\Java\jdk1.5.0.
  + Set CLASSPATH to C:\Program Files\Java\jdk1.5.0\_20\jre\lib.
  + Set PATH to JRE bin, e.g. C:\Program Files\Java\jre1.5.0\_20\bin.
* Install Database
* Install Database drivers: you should not worry about this part.
  + Most of the Database vendors are supplying appropriate JDBC drivers along with Database installation.
  + The latest JDK includes a JDBC-ODBC Bridge driver that makes most Open Database Connectivity (ODBC) drivers available to programmers using the JDBC API.
* Set database Credentials: When you install any of the above database, its administrator ID is set to root and gives provision to set a password of your choice. Using root ID and password you can either create another user ID and password, or you can use root ID and password for your JDBC application.

JDBC drivers : JDBC drivers implement the defined interfaces in the JDBC API, for interacting with database server.

The Java.sql package that ships with JDK, contains various classes with their behaviours defined and their actual implementaions are done in third-party drivers. Third party vendors implements the java.sql.Driver interface in their database driver.

There are four types of JDBC-Drivers:

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates.

* Type 1 driver: a JDBC bridge is used to access ODBC drivers installed on each client machine.
  + Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.
  + When Java first came out, most databases only supported ODBC access that's why these drivers were required.
* Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database.
  + These drivers are provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge.
  + The vendor-specific driver must be installed on each client machine.
  + If we change the Database, we have to change the native API
  + EX: Oracle Call Interface (OCI) driver
* Type 3 driver : The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.
  + It requires no code installed on the client and
  + A single driver can provide access to multiple databases.
* Type 4 driver : a pure Java-based driver communicates directly with the vendor's database through socket connection.
  + Usually provided by the vendor itself.
  + you don't need to install special software on the client or server.
  + Further, these drivers can be downloaded dynamically.
  + MySQL's Connector/J driver is a Type 4 driver.

If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver, otherwise preferred driver type is 4.

Steps involved establishing a JDBC connection

* Import JDBC Packages
  + import java.sql.\*
* Register JDBC Driver : Registering the driver is the process by which the Oracle driver's class file is loaded into the memory, so it can be utilized as an implementation of the JDBC interfaces. You can register a driver in one of two ways.
  + Class.forName() : dynamically load the driver's class file into memory, which automatically registers it. This allows you to make the driver registration configurable and portable.
    - Class.forName("oracle.jdbc.driver.OracleDriver");
  + DriverManager.registerDriver() - use the static method registerDriver() if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.
    - Driver myDriver = new oracle.jdbc.driver.OracleDriver();
    - DriverManager.registerDriver( myDriver );
* Database URL Formulation : A database URL is an address that points to your database.
  + MySQL # com.mysql.jdbc.Driver # **jdbc:mysql**://hostname/ databaseName
  + ORACLE # oracle.jdbc.driver.OracleDriver **# jdbc:oracle:thin**:@hostname:port Number:databaseName
  + DB2 # COM.ibm.db2.jdbc.net.DB2Driver # **jdbc:db2:**hostname:port Number/databaseName
  + Sybase # com.sybase.jdbc.SybDriver # **jdbc:sybase:Tds**:hostname: port Number/databaseName

All the highlighted part in URL format is static. you need to change only the remaining part as per your database setup.

* Create Connection Object : call getConnection() method with appropriate username and password to establish actual database connection.
  + Connection conn = DriverManager.getConnection(URL, USER, PASS);

Closing the connection:

At the end of your JDBC program, it is required explicitly to close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on the garbage collection, especially in database programming, is a very poor programming practice. You should make a habit of always closing the connection with the close() method associated with connection object. To ensure that a connection is closed you should call close() method inside a ‘finally’ block.

conn.close();

Explicitly closing a connection conserves DBMS resources, which will make your database administrator happy.

**Transaction:**

By default, JDBC Connection is in auto-commit mode. Every SQL statement is committed to the database upon its completion.

There are three reasons for turning off the auto-commit and manage your own transactions:

* To increase performance,
* To maintain the integrity of business processes,
* To use distributed transactions.

If you pass a boolean false to **Connection** object's setAutoCommit() method, you turn off auto-commit. You can pass a boolean true to turn it back on again.

conn.setAutoCommit(false);

Once you are done with your changes and you want to commit the changes then call commit() method on connection object.

conn.commit( );

To roll back updates to the database

conn.rollback( );

**Savepoints** : an additional transactional control

When you set a savepoint you define a logical rollback point within a transaction.

If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

The Connection object has two new methods to manage savepoints:

* setSavepoint(String savepointName): Defines a new savepoint.
  + It also returns a Savepoint object.
* releaseSavepoint(Savepoint savepointName): Deletes a savepoint.
  + Notice that it requires a Savepoint object as a parameter.
  + This object is usually a savepoint generated by the setSavepoint() method.

**Batch Processing**

Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.

Adv : improving performance, as it reduces the amount of communication overhead.

Batch Processing features are supported in Statement, PreparedStatement and CallableStatement by following methods -

* addBatch() : used to add individual statements to the batch
* executeBatch() : used to start the execution of all the statements grouped together.
  + returns an array of integers, and each element of the array represents the update count for the respective update statement.
* clearBatch() : removes all the statements from a batch which you added with the addBatch() method.
  + However, you cannot selectively choose which statement to remove.

JDBC drivers are not required to support this feature. **DatabaseMetaData.supportsBatchUpdates**() method returns true if JDBC driver supports this feature

Steps to use Batch Processing with Statment Object:

* Create a Statement object.
  + Statement stmt = conn.createStatement();
* Set auto-commit to false.
  + conn.setAutoCommit(false);
* Add as many as SQL statements you like into batch using statement object.
  + String SQL = "INSERT INTO Employees (id, name) VALUES(200,'Zia')";
  + stmt.addBatch(SQL);
  + stmt.addBatch(SQL1);
* Execute all the SQL statements on statement object.
  + int[] count = stmt.executeBatch();
* Finally, commit all the changes using commit() method.
  + conn.commit();

Batch Processing with PrepareStatement Object:

* Create SQL statements with placeholders.
  + String SQL = "INSERT INTO Employees (id, name) VALUES(?, ?)";
* Create PrepareStatement object using either prepareStatement() methods.
  + PreparedStatemen pstmt = conn.prepareStatement(**SQL**);
* Set auto-commit to false using setAutoCommit().
  + conn.setAutoCommit(false);
* Add as many as SQL statements you like into batch using addBatch() method on created statement object.
  + pstmt.setInt( 1, 100 );
  + pstmt.setString( 2, "Samrat" );
  + **pstmt.addBatch();**
  + pstmt.setInt( 1, 200 );
  + pstmt.setString( 2, "Virat" );
  + **pstmt.addBatch();**
* Execute all the SQL statements using executeBatch() method on created statement object.
  + int[] count = stmt.executeBatch();
* Finally, commit all the changes using commit() method.
  + conn.commit();