JDBC

**Java database connectivity** (JDBC) is the Java Software specification of a standard application programming interface (API) that allows Java programs to access database management systems. The JDBC API consists of a set of interfaces and classes written in the Java programming language.

**Data base over a File system**

1) Every project has frontend and backend end. The front end contains applications logics order, presentation logic, business logic and persistence logic.

2) A Backend is a repository of data it means Data Storage.

3) Initially we used a file as a backend for persisting the data.

4) A **persistence** data means, a data which has more life time then its process we also call this persistence data as a permanent data.

5) The logic which is used to make same data has persistence data is called **persistence** **logic**.

**With files we got some disadvantages**

1. Files are complex to work.
2. Files are less secure.
3. A file doesn’t have any query language support.
4. A file doesn’t support relationships between the data etc.

In order to overcome this drawbacks of file management system we entered into data base management system.

**What is the different between Driver class and Driver Software?**

A driver software contains a group of implementation classes, to access all the implementation classes of driver software, there is an entry point class called Driver Class.

A driver class provide a reference to an object of next class (connection) and it will provide an object another class etc like that all classes of driver software are accessed.

In other words a driver class is define has, it is a class which is implemented from Driver Interface given by sun.

In JDBC with the name “***Driver”*** we are calling Following 3 words

**i) Driver Software.**

**ii) Driver Class.**

**iii)Driver Interface.**

Driver classes are provided by vendors, which part of driver software and it is implemented from Driver Interface.

Note:- In JDBC there are two types of vendors

i) DB Vendors.

ii) A third party vendors.

A DB vendors means a company which created a driver software with having own DB software.

A third party vendors means, a company which provide Driver software but they don’t have own databases software.

What is an **API**?

An application-programming interface (API) is a set of programming instructions and standards for accessing another software application. Or An API is the collection of some class , interface and related files, which are designed to do certain operations.

In java API’s are nothing but the jar files.

What is **JAR (java Archive )**?

A jar file is a zip file which contains any kind of files, but they usually contain class files and supporting configuration files (properties), graphics and other data files needed by the application.

JAR file is the compressed file format. You can store many files in a JAR file. JAR stands for the **Java Archive**. This file format is used to distribute a set of java classes. This file helps you to reduce the file size and collect many file in one by compressing files.

How to **create JAR** file?

C:\vikas> jar –cvf ‘name.jar’ file1.class file2.class ......

C:\vikas> jar –cvf ‘name.jar’ \*.class (includes all the files with .class extension)

C:\vikas> jar –cvf ‘name.jar’ \*.\* (includes all the files with any extension)

C:\Good>jar -cvf kushi.jar \*.class

added manifest

adding: Apple.class(in = 412) (out= 291)(deflated 29%)

adding: Orange.class(in = 414) (out= 291)(deflated 29%)

adding: TakeObject.class(in = 576) (out= 382)(deflated 33

adding: Vikas.class(in = 292) (out= 224)(deflated 23%)

C:\Good>

Kushi.jar file is created. Now you can use this jar file has an API to any of your project, than in your project u can make use of all the features of Apple , Orange , TakeObject and Vikas class features.

 c' option indicates that you want to create a new JAR file;

 'v' option asks for *verbose* mode, which displays information messages;

 'f' specifies that the output go to a file specified in *jarfile* (instead of the standard output by default). Option 'f' and *jarfile* are a pair.

How to create **executable JAR** file?

1. Creating Executable JAR File By entry point

**easy way to create executable jars in Java**, as you need not have to create manifest file explicitly and it will be create by jar command itself along with Main-Class entry. What you need to provide is a new jar option "-e" and you main class name while running jar command. here is **example of jar command** with entry option

**jar -cvfe fileName.jar MainClassName MainClassName.class Other.class...**

C:\Good>jar -cvfe viku.jar Vikas Vikas.class Apple.class Orange.class TakeObject.class

added manifest

adding: Vikas.class(in = 292) (out= 224)(deflated 23%)

adding: Apple.class(in = 412) (out= 291)(deflated 29%)

adding: Orange.class(in = 414) (out= 291)(deflated 29%)

adding: TakeObject.class(in = 576) (out= 382)(deflated 33%)

C:\Good>

**jar -e for entry point** and entry point or main class name should come after jar file name and before directory or file needs to be included in JAR

Than run the jar file

C:\Good>java -jar viku.jar

Its so sweet to eat an orange.....!

C:\Good>

(note : above output is with respect to my program so u need not to wory)

(note : this also works C:\Good>jar -cvfe viku.jar Vikas \*.class)

2) Creating Executable JAR File Example with External Manifest

To **create an executable JAR in Java**, you need to provide a manifest file and include your Main Class in Manifest

Create MANIFEST.MF file by using any text editor e.g. notepad in windows or Vim in Unix and add following entry in file, remember last line must end with either new line or carriage return

Manifest-version: 1.0  
Main-Class: Vikas

Important thing to remember is that we need to specified full classified class name here. suppose if our main class was inside com/jspiders/Vikas than we should have to specify com. jspiders. Vikas here, don't put .class extension here its not required

Than execute the following command

**jar -cvfm fileName.jar MANIFEST.MF file1.class file2.class ...**

here **-m is used for including manifest file** and remember specify name of manifest file after jar name. now you have an executable jar file in java which you run by command specified earlier

C:\Good>jar -cvfm kushi.jar MANIFEST.MF \*.class

added manifest

adding: Apple.class(in = 412) (out= 291)(deflated 29%)

adding: Orange.class(in = 414) (out= 291)(deflated 29%)

adding: TakeObject.class(in = 576) (out= 382)(deflated 33%)

adding: Vikas.class(in = 292) (out= 224)(deflated 23%)

C:\Good>

Than execute the jar file by using the command

C:\Good>java -jar kushi.jar

Its so sweet to eat an orange.....!

C:\Good>

(note : above output is with respect to my program so u need not to wory)

What are **MANIFEST** file?

The manifest is a special file that can contain information about the files packaged in a JAR file. By tailoring this "meta" information that the manifest contains, you enable the JAR file to serve a variety of purposes.

Manifest file is mainly known for contains the name of the class that holds the main function, among various classes in the package...

When you create a JAR file, it automatically receives a default manifest file. There can be only one manifest file in an archive, and it always has the pathname

META-INF/MANIFEST.MF

What are **javadocs**?

**Javadoc** is a documentation generator from Oracle Corporation for generating API documentation in HTML format from Java source code. The HTML format is used to add the convenience of being able to hyperlink related documents together

Creating the javadoc

**C:\vikas>Javadoc filename.java**

**What is JDBC?**

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks commonly associated with database usage

* Making a connection to a database
* Creating SQL or MySQL statements
* Executing that SQL or MySQL queries in the database
* Viewing & Modifying the resulting records

## JDBC is an API it helps us to achive connectivity between java and database

## JDBC is the one and only one API which helps us to interact with the database

## Advantages

## JDBC api is built upon java it inherits the advantages of java

## JDBC is database independent

## Using JDBC we can interact with multiple database

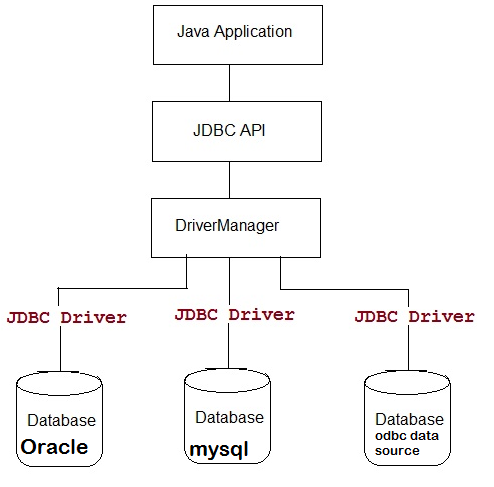
## The performance of JDBC is increased by PrepareStatement and CallableStatement

## JDBC Architecture

* **JDBC API:** Provides the application-to-JDBC Manager connection.
* **JDBC Driver API:** Supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide the connectivity to the required databases.

The JDBC driver manager is a traditional backbone of the JDBC architecture, which specifies objects for connecting Java applications to a JDBC driver



All the JDBC API’s are present in two package

java.sql

javax.sql

**java.sql** includes classes and interface to perform all the JDBC operations.

**javax.sql** this package is known has extension API. It provides classes and interface to accesses server side data.

**Driver**

Driver is the one which contains a set of code which is used to communicate between the two systems. The Driver works like translator. Eg : Printer Driver , JDBC driver and etc

Driver can be written in any language like C,C++,java

**JDBC Driver**

It is the addition software which is used to interact with data base

Is the one which is used to communicate between the java program and the DB, it also work has a translator between them.

The Drivers are provided by DB vendor & they are DB dependent

The Driver is a concrete class inside the driver.jar file, which implements the java.sql.Driver interface.

This interface present in jdbc api & every JDBC driver provider has to implement this interface

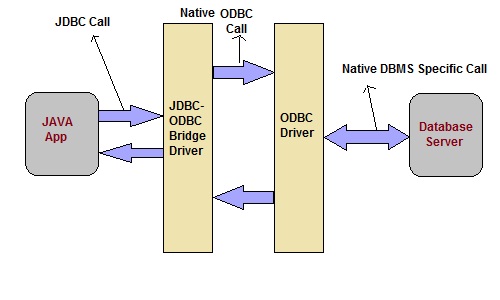
The driver helps us to establish DB connection transfer the DB query & results between java program and DB

A Java program that uses the **JDBC** API loads the specified **driver** for a particular DBMS before it actually connects to a database. The **JDBC** DriverManager class then sends all **JDBC** API calls to the loaded **driver**. The four types of **JDBC drivers**

There are different types of JDBC drivers

* **Type1** JDBC-ODBC bridge driver
* **Type2** Native-API, partly Java driver
* **Type3** JDBC-Net, pure Java driver
* **Type4** Native-protocol, pure Java driver

**Type-1 : JDBC-ODBC bridge driver**

****

The driver converts JDBC method calls into ODBC function calls. The driver is platform-dependent as it makes use of ODBC which in turn depends on native libraries of the underlying [operating system](http://en.wikipedia.org/wiki/Operating_system) the [JVM](http://en.wikipedia.org/wiki/JVM) is running upon. Also, use of this driver leads to other installation dependencies; for example, ODBC must be installed on the computer having the driver and the database must support an ODBC driver. Sun provides a JDBC-ODBC Bridge driver: **sun.jdbc.odbc.JdbcOdbcDriver**. This driver is native code and not Java, and is closed source.

### Advantages

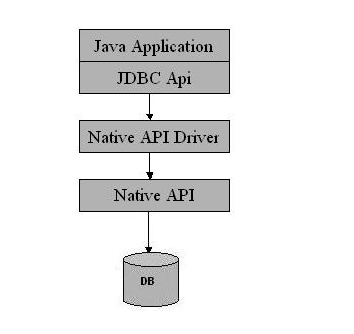
Almost any database for which an ODBC driver is installed can be accessed, and data can be retrieved.

### Disadvantages

* Performance overhead since the calls have to go through the jdbc Overhead bridge to the ODBC driver, then to the native db connectivity interface (thus may be slower than other types of drivers).
* The ODBC driver needs to be installed on the client machine.
* Not suitable for applets, because the ODBC driver needs to be installed on the client.

**Type-2 : Native-API, partly Java driver**

The distinctive characteristic of type 2 jdbc drivers are that Type 2 drivers convert JDBC calls into database-specific calls i.e. this driver is specific to a particular database. Some distinctive characteristic of type 2 jdbc drivers are shown below. Example: Oracle will have oracle native api.



##### **Advantages**:

The distinctive characteristic of type 2 jdbc drivers are that they are typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type-1 and also it uses Native api which is Database specific.

##### **Disadvantages**:

1. Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.

2. Like Type 1 drivers, it's not written in Java Language which forms a portability issue.

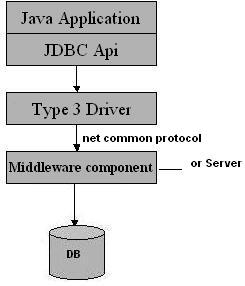
3. If we change the Database we have to change the native api as it is specific to a database

4. Mostly obsolete now

5. Usually not thread safe.

**Type-3 : JDBC-Net, pure Java driver**

The JDBC type 3 driver, also known as the Pure Java Driver for Database **Middleware**, is a database driver implementation which makes use of a [middle tier](http://en.wikipedia.org/wiki/Middle_tier) between the calling program and the database. The middle-tier ([application server](http://en.wikipedia.org/wiki/Application_server)) converts [JDBC](http://en.wikipedia.org/wiki/JDBC) calls directly or indirectly into the vendor-specific [database](http://en.wikipedia.org/wiki/Database) protocol.



##### **Advantage** :

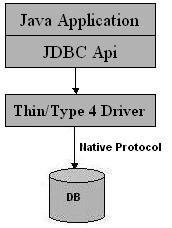
1. This driver is server-based, so there is no need for any vendor database library to be present on client machines.
2. This driver is fully written in Java and hence Portable. It is suitable for the web.
3. There are many opportunities to optimize portability, performance, and scalability.
4. The net protocol can be designed to make the client JDBC driver very small and fast to load.
5. The type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced system administration such as logging and auditing.
6. This driver is very flexible allows access to multiple databases using one driver.
7. They are the most efficient amongst all driver types.

##### **Disadvantage** :

1. It requires another server application to install and maintain.
2. Traversing the recordset may take longer, since the data comes through the backend server.

**Type-4 : Native-protocol, pure Java driver**

The JDBC type 4 driver, also known as the Direct to Database **Pure Java Driver**, is a database driver implementation that converts [JDBC](http://en.wikipedia.org/wiki/JDBC) calls directly into a vendor-specific [database](http://en.wikipedia.org/wiki/Database) protocol.



###### **Advantage**

1. The major benefit of using a type 4 jdbc drivers are that they are completely written in Java to achieve platform independence .
2. Number of translation layers is very less i.e. type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good.
3. You don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

##### **Disadvantage**

1. With type 4 drivers, the user needs a different driver for each database.

**Steps to establish the JDBC connections**

1. Load and register the Driver
2. Establish the connection
3. Create statement
4. Execute the query
5. Close the connection

**Step 1 - Load and register the Driver**

In this step we need to load the Driver class of the particular Data base. You need to give a fully qualified name of the Driver class to the forName( ) method, which is the static method of the class **Class** which is present in **java.lang** package. This will throw the checked exception that is ClassNotFoundException , we need to handle it.

Class.forName( “ fully Qualified Path Name ”);

**MySql**  Class.forName(“**com.mysql.jdbc.Driver**”);

**Oracle** Class.forName(“**oracle.jdbc.driver.OracleDriver**”);

**ODBC** Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Other way to load and register the driver

DriverManager.registerDriver(new com.mysql.jdbc.Driver ());

DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

Or

import com.mysql.jdbc.Driver;

Dirver d = new Driver();

DriverManager.registerDriver(d);

Note: Diver() constructor and registerDriver() method both throws **SqlException**.

**DriverManager**

DriverManager provides a basic service for managing a set of JDBC drivers. Is the helper class, present in **java.sql** package . It acts has an interface between the program and the Driver. DriverManager is the factory class. When we load the JDBC Driver class, it registers itself to the DriverManager. It keeps the track of all the Drivers registered, it has many methods like

registerDriver( Driver class Object)🡪used to register driver class object

deregisterDriver(Driver class Object)🡪used to deregister the give driver

getConnection( ..... ) methods🡪used to establish the connection

**Step 2 - Establish the Connection**

**DriverManager**

* Its a part of JDBC api and has the name implies it manages the drivers. This class helps us to establish connection to database
* Its a concrete class it has only one constructor which is private default in nature, hence this class cannot be instantiated or inherited. This class have many public static in nature.
* This class requires two information to establish connection
  + - 1. DB URL
      2. Driver Class ( which is present in the Driver jar file)

In this step we are going to connect to the DB, to do this we are going to use the getConnection( ) method of the DriverManager class, which is present in **java.sql** package.

The getConnection( ) method is the overloaded static method of DriverManager class. It throws the checked exception that is the **SQLException**. The return type of getConnection( ) is the Connection type (implementation Object), which is also present in the java.sql package

getConnection(Srting url);

getConnection(String url , String username , String password);

getConnection(String url , Properties file);

**MySql** getConnection(“jdbc:mysql://localhost:3306/dataBaseName”, “username”, “password”);

**Oracel** getConnection(“jdbc:oracle:thin:@localhost:5121:XE”, “username”, “password”);

**ODBC** getConnection("jdbc:odbc:dataBaseName");

Note : any class which implements an interface is called has implementation class, the object of implementation class called has implementation Object.

**Step 3 - Create Statement (Establish the Statement)**

There are three types of statement

* + - * Statement
      * PreparedStatement
      * CallableStatement

Using the connection object reference, we can create the above statement objects by calling the corresponding methods (has shown below).

To create Statement we use **createSatement( )** method. Return type of this method is Statement type object. We genrally use Statement type object to execute the static query.

To create PreparedStatement we use **preparedSatement(String sql )** method. Return type of this method is PreparedStatement type Object. We genrally use preparedStatement type object to execute the dynamic queries.

To create CallableStatement we use **prepareCall(String sql )** method. Return type of this method is CallableStatement type Object. We make use of this object to execute the stored procedures.

**Step 4 - Execute the sql query**

This is the step were we are going to send our sql queries and ask them to execute in the DB, there are many methods that are used to execute the query but one thing is that we must be knowing where and which we need to use when.

* **boolean execute( )** : This method return true, if we execute Select statement. This return false if we execute non-select statements.
* **int executeUpdate(String SQL)** : Returns the numbers of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
* **ResultSet executeQuery(String SQL)** : Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

**Step 5 - Close the connection**

After performing all the operations its a good practice to close the connection by calling the close( ) method. It also throws the checked SQLException so it need to be handled. Its good practice to execute this statement inside the finally block.

**Difference between Statements and PreparedStatements**

When ever you execute the query these are the steps the compiler of Db is going to perform

Step 1 : Parse the incoming SQL query  
Step 2 : Compile the SQL query  
Step 3 : Plan/optimize the data acquisition path  
Step 4 : Execute the optimized query / acquire and return data

|  |  |
| --- | --- |
| **Statement** | **PreparedStatement** |
| These are not pre compiled statements , each time when u execute the query get compiled again and again | These are pre compiled statements, hear the query gets compiled once and it can execute has many number of times we need to execute |
| Statement will be used for executing static SQL statements and it can't accept input parameters | PreparedStatement will be used for executing SQL statements many times dynamically. It will accept input parameters. |
| Best for DDL commands | Best for DML commands |
| These are slow in execution | These are fast in execution |
| A Statement will always proceed through the four steps above for each SQL query sent to the database | Prepared Statement pre-executes steps (1) - (3) in the execution process above. Thus, when creating a Prepared Statement some pre-optimization is performed immediately. The effect is to lessen the load on the database engine at execution time |
| It does not provide any methods , to insert the parameters | PreparedStatement provides different types of setter methods to set the input parameters for the query |
| If we use statement type object than we need to pass the sql query into execute methods.  **Example:**  **stm.execute(sql);**  **stm.executeQuery(sql);**  **stm.executeUpdate(sql);**  **stm.addBatch(sql)** | If we use PrepareStatement type object than we need not to pass any sql query into execte methods.  **Example:**  **pstm.execute();**  **pstm.executeQuery();**  **pstm.executeUpdate();**  **pstm.addBatch();** |

**Difference between execute( ) , executeUpdate( ) and executeQuery( )**

**execute**(String query) is used to execute any SQL query and it returns TRUE if the result is an ResultSet such as running Select queries. The output is FALSE when there is no ResultSet object such as running Insert or Update queries. We can use getResultSet() to get the ResultSet (When Select statement is executed) and getUpdateCount() method to retrieve the update count(When DML statements are executed).

Statement **executeQuery**(String query) is used to execute Select queries and returns the ResultSet. ResultSet returned is never null even if there are no records matching the query. When executing select queries we should use executeQuery method so that if someone tries to execute insert/update statement it will throw java.sql.SQLException with message “executeQuery method can not be used for update”.

Statement **executeUpdate**(String query) is used to execute Insert/Update/Delete (DML) statements or DDL statements that returns nothing. The output is int and equals to the row count for SQL Data Manipulation Language (DML) statements. For DDL statements, the output is 0.

**Callable Statement:**

1. Callable Statement is an interface extends from PreparedStatement and given in java.sql
2. Callable statement has all the benefits of preparedStatement and also callableStatement has one additional features when compared with PrepareStatement that is A Callable Statement can call pl/sql procedure or a function of a database apart from executing sql commands.
3. In JDBC , it is not possible to call a procedure or a function using a Statement or PreparedStatement Interface.
4. From a java application , it is not possible to call a trigger explicitly database trigger will be fired automatically at database side and cannot be called from java.

**Syntax:**

For Executing SQL Command :

CallableStatement cstmt=con.prepareCall(“SQL Command”);

For Executing Procedure or Function :

CallableStatement cstmt=con.prepareCall({call ProcedureName(args) or funtiond(args)}”);

1. Callable Statement Interface can be used for both executing SQL command and also for calling procedure and function but we use callable Statement mostly for calling procedure and function.
2. We can store a callable statement object in PreparedStatement because CallableStatement is derived from PreparedStatement.

PreparedStatement pstmt=con.prepareCall(“SQL Command”);

1. The difference between a procedure and function in programming language like C or C++ etc is a Procedure doesn’t return any value and function returns a single value.
2. In programming language, if a function return type is void then we call it has a procedure.
3. In database programming the difference between a Procedure and A function is procedure can return Zero or more values but a function exactly return only one value.

**Transaction Management In JDBC:**

* **A** transaction is a group of operation’s made as a unit and produces a single outcome called either success or fail.
* **A** transaction becomes success, if all operation in the group is success.
* **A** transaction becomes failed, if any one operation in a group is failed.
* **A** group of operation’s which follows either all or nothing principle is called a transaction.
* **In** a transaction, if one operation is failed then all operation are cancelled.
* A transaction becomes success, if and only if all operation in the group are successfully executed.

***If all is well Success***

**I) Reserve Seat**

**II) Payment**

**III) Issue Ticket**

Purchase ticket

***If something is wrong fail***

**Types of Transaction:**

1. Local Transaction
2. Global Transaction / Distributed Transaction

A local transaction means a transaction whose operation are executed against a single Database

A global or Distributed Transaction means a transaction whose operations are executed on more than one database.

JDBC technology supports only a local transaction, to perform global transaction in java we need to use EJB technology (out Dated) or Spring Framework.

JDBC Transaction Management:

* While executing any database operation like insert, update or delete operation on a database then the operation will be permanently executed on a database. If once it is executed it cannot be rollback.
* The reason why a command is permanently executed on java is because AutoCommit mode is enabled (true) in java by default, if AutoCommit mode is in enabled state then the command will be permanently executed and cannot be cancelled after executing.
* In transaction concept, if one operation failed we need to cancel all operation which are already executed on a database within that transaction so to perform transaction management in JDBC, first of all we need to disabled AutoCommit mode.
* To make a group of operation has a transaction; we need to put the operation in the try block.
* If exception occurred in try block operation then we need to cancel by calling rollback ().
* If no exception is occurred in a set of try block operation then we need to commit the transaction by calling commit () method.
* To disabled AutoCommit mode we need to call a method **SetAutoCommit(false).**
* Finally the following 3 methods provided by connection Interface are used to perform transaction management in JDBC

con.setAutoCommit()

con**.**commit()

con.rollback()

**Save points**

It is used to have multiple logical units under one transaction. Using the save point we can roll-back only part of the transaction instead of roll-backing the entire transaction.

|  |  |
| --- | --- |
| **Method** | **Task** |
| Savepoint **setSavepoint**(String savepointName) | This method is used to create the save point. This method also return the savepoint object |
| void **releaseSavepoint**(Savepoint ref) | This method is used to delete the save point, this method requires the save point object (reference) to delete the save point |
| void **rollback**(Savepoint ref) | This method rollback the changes until the given save point. It requires the savepoint object(reference) to rollback. |

**Batch Processing:-**

* From a java application, if we run some SQL operation on a database then the SQL command are executed as one by one on to the database.
* If a java application contains more number of SQL operation contains more number of SQL operation then the network roundtrips between an application and database will be increased so that an application performance will be decreased.
* In order to overcome the above drawback or problem we got batch processing technique.
* In batch processing, multiple SQL operation are made as one batch and then the entire batch will be at a time transferred on to the database and execute. It means ‘n’ number of roundtrips are reduced to a single Round Trip so that performance of an application will be increased
* To do batch processing we got the following two methods in StatementInterface.
  + 1. Void addbatch(“SQL Command”);
    2. Int[] executeBatch();
* addbatch () is a method for constructing a batch by grouping all SQL commands.
* While making a batch, a batch doesn’t allow select operation. It means a batch should contain only non select operation.
* Execute batch method will transfer the batch and execute operation on database and returns result of the batch as an “int[]”.
* While execute the batch if any error occurred in the middle then this execute batch methods throws “BatchUpdateException”
* While executing a batch if an exception occurs in the middle then the next part of the Batch will not execute and previous part of the batch will not be cancelled.
* If we want to cancel the total batch operators when an Exception is occurred then we need to apply or combine transaction management with Batch Processing.

**Data Base URL**

Data base uniform locator is used to uniquely identify the database in the network. The structure of DB URL is

<protocol>:<sub-protocol>:<sub-name>

**Protocol** - with respect to JDBC its always **jdbc**

**Sub-protocol –** its the name of the data-base connectivity mechanism. This information will be present in the database manual.

**Sub-name**

It contains

* Host name
* Port number
* Data base name
* Ser name and password

**Overloading of getConnection( ) method of DriverManager class**

**DriverManager.getConnection(String url )**

If you want to use the DriverManager class to create connection objects, you need to know how to make a connection URL that provides access information to the MySQL server. The MySQL connection URL has the following format.

***jdbc:mysql://[host][:port]/[database][?property1][=value1]...***

***host*** *- The host name where MySQL server is running.*

*Default is 127.0.0.1 - the IP address of localhost.*

***port*** *- The port number where MySQL is listening for connection.*

*Default is 3306.*

***Database*** *- The name of an existing database on MySQL server.*

*If not specified, the connection starts no current database.*

***Property*** *- The name of a supported connection properties.*

*"user" and "password" are 2 most important properties.*

***Value*** *- The value for the specified connection property.*

**Example:**

**URL = jdbc:mysql://localhost:3306/school?user=root&password=root**

DriverManager.*getConnection*("jdbc:mysql://localhost:3306/school?user=root&password=imking");

**DriverManager.getConnection(String url , String user , String password )**

This getConnection excepts 3 parameters, the path or the url (jdbc:mysql://localhost:3306 ) of the DB than username and password what you had given while installing in your system. ( the path or url can also contain the data-base name) .

**DriverManager.getConnection(String url , Properties pro);**

This is the connection which can be established by using **Properties** class and also by using the **.properties** file.

**Using Properties class**

Properties is a class which is a Map, it is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String .

**Object setProperty(String key, String value)**  
Associates value with key. Returns the previous value associated with key, or returns null if no such association exists.

**String getProperty(String key)**  
Returns the value associated with key

Properties props = **new** Properties();

props.setProperty("user", "root");

props.setProperty("password", "imking");

Connection con = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/school", props);

**Using Properties file**

.**properties** is a **file** extension for **files** mainly used in **Java** related technologies to store the configurable parameters of an application. They can also be used for storing strings for Internationalization and localization; these are known as **Property** Resource Bundles.

.properties file is a simple collection of key-value pairs that can be parsed by the [java.util.Properties](http://java.sun.com/javase/6/docs/api/java/util/Properties.html) class.

Properties files are widely used for many purposes in all kinds of Java applications, often to store configuration or localization data.

Create one file with the **.properties** extension, hear i have created the properties file has king.properties, and i have stored these parameters inside it.

**user:**root

**password:**imking

Later you have to read this file, by using the Properties class, has shown below

Properties props = **new** Properties();

props.load(**new** FileInputStream("king.properties"));

Connection con = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/school",props);

**Use of properties file**

If you put **Driver name , path , user , password** in a class or interface you would need to recompile not only that class, but the entire application because the constants are probably inlined wherever they are used. By putting them in property files (or any other form of configuration files, e.g. XML) you can modify the file and restart your application, and all code will use the new values. This also allows non-programmers to modify them if needed.

Generally we are going to put database related details into our properties file and we accesses them, whenever we need them.

**Example:**

**King.properties** file contains the below 3 lines

driver : com.mysql.jdbc.Driver

user : root

password : imking

The java code to establish the connection using the information stored in properties file we have to write the code has shown below.

Properties props = **new** Properties();

props.load(**new** FileInputStream("king.properties"));

String driver = props.getProperty("driver");

String user = props.getProperty("user");

String pass = props.getProperty("password");

Class.*forName*(driver);

Connection con = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/school",user,pass);

**ResultSet**

A ResultSet object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The next method moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set.

|  |  |
| --- | --- |
| **1) public boolean next();** | is used to move the cursor to the one row next from the current position. |
| **2) public boolean previous();** | is used to move the cursor to the one row previous from the current position. |
| **3) public boolean first();** | is used to move the cursor to the first row in result set object. |
| **4) public boolean last();** | is used to move the cursor to the last row in result set object. |
| **5) public boolean absolute(int row);** | is used to move the cursor to the specified row number in the ResultSet object. |
|  |  |
| **6) public int getInt(int columnIndex);** | is used to return the data of specified column index of the current row as int. |
| **7) public int getInt(String columnName);** | is used to return the data of specified column name of the current row as int. |
| **8) public String getString(int columnIndex);** | is used to return the data of specified column index of the current row as String. |
| **9) public String getString(String columnName);** | is used to return the data of specified column name |

# Meta data

# 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 **getColumnTypeName(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. |

### Creating object of ResultSetMetaData:

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

public ResultSetMetaData **getMetaData( )** throws SQLException

**Database MetaData:**

1. This MetaData Interface is used to retrieve Metadata of a database with combination of JDBC driver.
2. To get database Metadata object, we need to call getMetaData() method given by Connection Interface. **Syntax: DatabaseMetaData dbmd = con.getMetaData();**
3. We need to call methods of DatabaseMetaData to read MetaData from object.

|  |  |  |
| --- | --- | --- |
| Sl No | Method Name | Description |
| 1 | getDatabaseProductName() | It returns database product name |
| 2 | getDatabaseProductVersion() | It returns database product version |
| 3 | getDriverName() | It returns JDBC Driver name |
| 4 | getDriverVersion() | It returns Driver Version |
| 5 | getMaxColumnsInTable() | It returns max column allowed in table |
| 6 | getMaxColumnNameLength() | It returns max number of character allowed in a column name |
| 7 | getMaxTableNameLength() | It returns max number of character allowed in a table name |