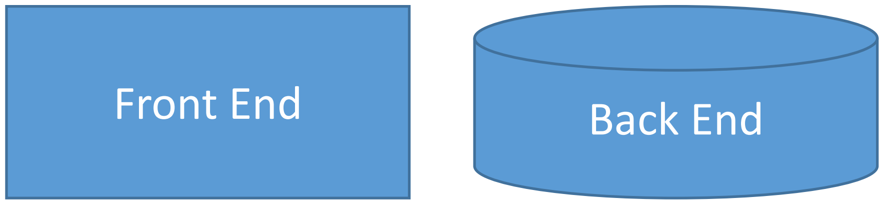
In software engineering, the terms "front end" and "back end" are distinctions which refer to the separation of concerns between a presentation layer and a data access layer respectively

The front end is an interface between the user and the back end. The front and back ends may be distributed amongst one or more systems.



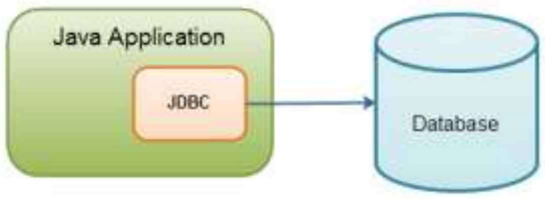
JDBC – Java Database Connectivity

Java Database Connectivity (JDBC) is an application program interface (API) specification for connecting programs written in Java to the data in popular databases. The application program interface lets you encode access request statements in Structured Query Language (SQL) that are then passed to the program that manages the database. It returns the results through a similar interface.

Although JDBC was designed specifically to provide a Java interface to relational databases, you may find that you need to write Java code to access non-relational databases as well

Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which defines how a client may access a database. It is part of the Java Standard Edition platform, from Oracle Corporation. It provides methods to query and update data in a database, and is oriented towards relational databases. A JDBC-to-ODBC bridge enables connections to any ODBC-accessible data source in the Java virtual machine (JVM) host environment.

The Java JDBC API enables Java applications to connect to relational databases via a standard API, so your Java applications become independent (almost) of the database the application uses.



History and implementation

Sun Microsystems released JDBC as part of Java Development Kit (JDK) 1.1 on February 19, 1997. Since then it has been part of the Java Platform, Standard Edition (Java SE). The JDBC classes are contained in the Java package java.sql and javax.sql.

Starting with version 3.1, JDBC has been developed under the Java Community Process. JSR 54 specifies JDBC 3.0 (included in J2SE 1.4), JSR 114 specifies the JDBC Rowset additions, and JSR 221 is the specification of JDBC 4.0 (included in Java SE 6).

JDBC 4.1, is specified by a maintenance release 1 of JSR 221 and is included in Java SE 7.

The latest version, JDBC 4.2, is specified by a maintenance release 2 of JSR 221 and is included in Java SE 8

ODBC

ODBC is Developed by Microsoft with collaboration with IBM

An ODBC driver uses the Open Database Connectivity (ODBC) interface by Microsoft that allows applications to access data in database management systems (DBMS) using SQL as a standard for accessing the data. Application end users can then add ODBC database drivers to link the application to their choice of DBMS. It supports DBMS Specific Drivers

Problems with ODBC

- Only connect with few databases only
- Not Vendor specific
- Some databases doesn’t support ODBC. They have their own Native Libraries written in C, C++ languages
- Using Vendor Native Libraries java communicates with the specific databases

Note: Java can also use ODBC to communicate with different databases

- But, being platform independent it depends on ODBC or Vendor Native Libraries
- The Java Application have to use Java Native Interface (JNI) to interact with Native Libraries and there are bugs in Native Libraries which may crash JVM as many Vendor Native Libraries are written in C, C++
- Sun Microsystem introduced JDBC, a common specification with different Databases.
- JDBC also uses SQL to communicate with database internally
- The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.
- 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.

Functionality

JDBC allows multiple implementations to exist and be used by the same application. The API provides a mechanism for dynamically loading the correct Java packages and registering them with the JDBC Driver Manager. The Driver Manager is used as a connection factory for creating JDBC connections.

JDBC connections support creating and executing statements. These may be update statements such as SQL's CREATE, INSERT, UPDATE and DELETE, or they may be query statements such as SELECT. Additionally, stored procedures may be invoked through a JDBC connection. JDBC represents statements using one of the following classes:

- **Statement** – the statement is sent to the database server each and every time.
- **PreparedStatement** – the statement is cached and then the execution path is pre-determined on the database server allowing it to be executed multiple times in an efficient manner.
- **CallableStatement** – used for executing stored procedures on the database.

Update statements such as INSERT, UPDATE and DELETE return an update count that indicates how many rows were affected in the database. These statements do not return any other information.

Query statements return a JDBC row result set. The row result set is used to walk over the result set. Individual columns in a row are retrieved either by name or by column number. There may be any number of rows in the result set. The row result set has metadata that describes the names of the columns and their types.

There is an extension to the basic JDBC API in the javax.sql.

JDBC connections are often managed via a connection pool rather than obtained directly from the driver.

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

- 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. Java can be used to write different types of executables, such as −

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)

- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

JDBC Architecture

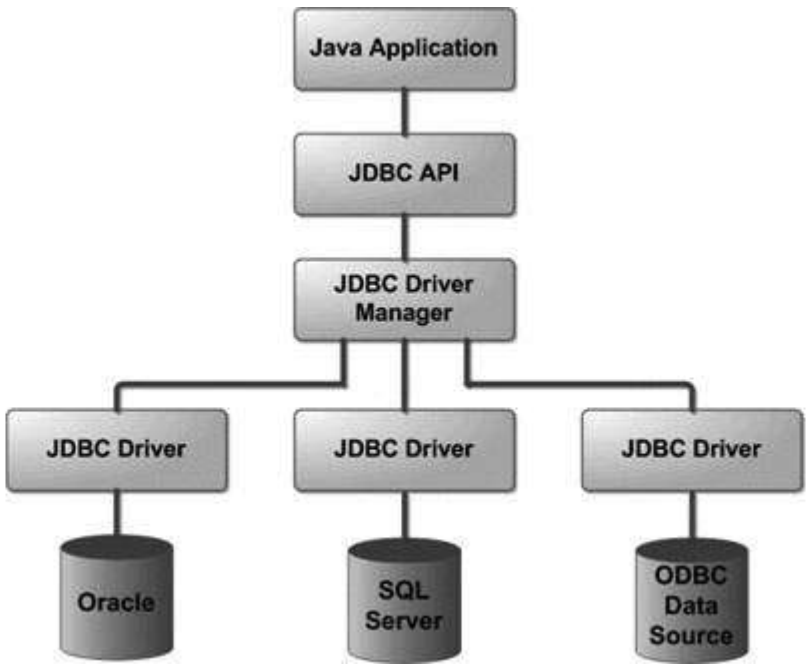
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

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

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.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application −



Common JDBC Components

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 sub protocol. 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. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **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.

JDBC Drivers

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

For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.

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.

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below −

1. Type I – (JDBC-ODBC Bridge Driver)
2. Type II – (Native-API, partly Java driver)

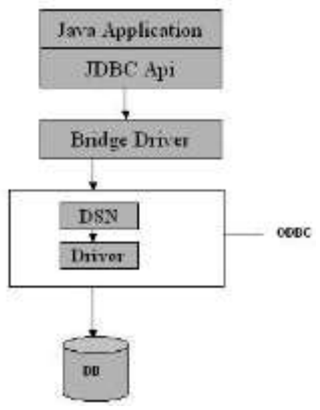
3. Type III – (JDBC-Net, Pure Java driver)
4. Type IV – (NATIVE-PROTOCOL, Pure Java driver)

Type I – (JDBC-ODBC Bridge Driver)

The JDBC type 1 driver, also known as the JDBC-ODBC bridge is a database driver implementation that employs the ODBC driver to connect to the database

In a 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.

The Type 1 driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver. ODBC is a generic API. The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available.



Advantages of Type I Driver

- Single driver implementation used to connect with different databases
- The JDBC-ODBC Bridge allows access to almost any database, since the database’s ODBC drivers are already available.
- It is Vendor independent driver

Disadvantages

- Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable
- Performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process. They are the slowest of all driver types
- The client system requires the ODBC Installation to use the driver
- Not good for the Web

Note: This type of driver is not recommended to use on production environment

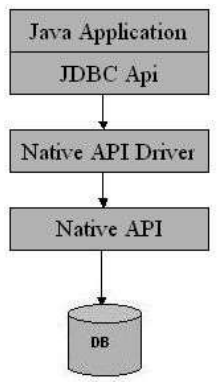
Type II – (Native-API, partly Java driver)

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically 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, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

- The type 2 driver is not written entirely in Java as it interfaces with non-Java code that makes the final database calls
- It Converts JDBC calls directly into database vendor specific calls
- The native API is developed in C/C++
- It converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS.

Note that, like the bridge driver, this style of driver requires that some binary code be loaded on each client machine



Advantage

- Better performance than Type 1 since no JDBC to ODBC translation is needed

Disadvantages

- Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet
- Like Type 1 drivers, it's not written in Java Language which forms a portability issue.
- If we change the Database we have to change the native api as it is specific to a database
- Usually not thread safe.
- DB Specific functions are executed on client JVM and any bug in this driver can crash JVM as Native Libraries are written in C/C++

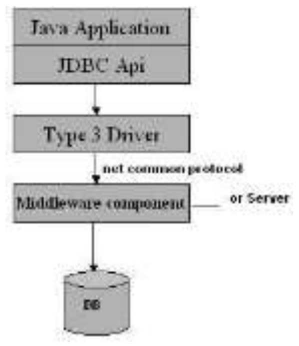
Type III – (JDBC-Net, Pure Java driver)

Type 3 database requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. If the middle-tier server can in turn use Type1, Type 2 or Type 4 drivers.

In a Type 3 driver, a three-tier approach is used to access databases. 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.

This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

- Supports 3-Tier Architecture
- Makes use of a middle-tier between the calling program and the database. The middle-tier (application server) converts JDBC calls directly or indirectly into the vendor-specific database protocol
- Type III are pure Java drivers and are auto downloadable



Advantage

- This driver is server-based, so there is no need for any vendor database library to be present on client machines.
- This driver is fully written in Java and hence Portable. It is suitable for the web.
- There are many opportunities to optimize portability, performance, and scalability.
- The net protocol can be designed to make the client JDBC driver very small and fast to load.
- 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
- This driver is very flexible allows access to multiple databases using one driver.
- They are the most efficient amongst all driver types.

Disadvantage

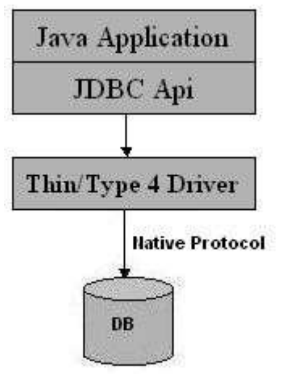
It requires another server application to install and maintain. Traversing the record set may take longer, since the data comes through the backend server.

Type IV – (NATIVE-PROTOCOL, Pure Java driver)

In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.

This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

- The type 4 driver is written completely in Java and is hence platform independent
- It is installed inside the Java Virtual Machine of the client
- It provides better performance over the type 1 and 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls
- Directly interacts with Database
- It Translates JDBC calls to DB Specific calls



Advantages

- Doesn’t need any plugin
- It is auto downloadable
- No need to install native libraries on client systems
- It doesn’t require any middleware or plugins to connect to database

Disadvantages

- It uses DB specific proprietary protocol, so depends on database vendor

Which Driver should be Used?

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 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 Connections

Import JDBC Packages

The **Import** statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code.

To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code –

```
import java.sql.* ;  // for standard JDBC programs
import java.math.* ; // for BigDecimal and BigInteger support
```

Register JDBC Driver/Load the JDBC Driver

You must register the driver in your program before you use it. 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 need to do this registration only once in your program. You can register a driver in one of two ways.

Approach I - Class.forName()

The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

The following example uses **Class.forName( )** to register the Oracle driver –

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

You can use **getInstance()** method to work around noncompliant JVMs, but then you'll have to code for two extra Exceptions as follows –

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
catch(IllegalAccessException ex) {
   System.out.println("Error: access problem while loading!");
   System.exit(2);
catch(InstantiationException ex) {
   System.out.println("Error: unable to instantiate driver!");
   System.exit(3);
}
```

Approach II - DriverManager.registerDriver()

The second approach you can use to register a driver, is to use the static **DriverManager.registerDriver()** method.

You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses registerDriver() to register the Oracle driver –

```
try {
   Driver myDriver = new oracle.jdbc.driver.OracleDriver();
   DriverManager.registerDriver( myDriver );
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

Database URL Formulation

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded DriverManager.getConnection() methods –

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)

Here each form requires a database **URL**. A database URL is an address that points to your database.

Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc :mysql:** //hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:** @localhost:1521:XE

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
Odbc	sun.jdbc.odbc.JdbcOdbcDriver	**jdbc:odbc:dsnname**

Create Connection Object

We have listed down three forms of **DriverManager.getConnection()** method to create a connection object.

Using a Database URL with a username and password

The most commonly used form of getConnection() requires you to pass a database URL, a *username*, and a *password*:

Assuming you are using Oracle's **thin** driver, you'll specify a host:port:databaseName value for the database portion of the URL.

If you have a host at TCP/IP address 192.0.0.1 with a host name of amrood, and your Oracle listener is configured to listen on port 1521, and your database name is EMP, then complete database URL would be –

```
jdbc:oracle:thin:@amrood:1521:EMP
```

Now you have to call getConnection() method with appropriate username and password to get a **Connection** object as follows –

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password"
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

Using Only a Database URL

A second form of the DriverManager.getConnection( ) method requires only a database URL –

```
DriverManager.getConnection(String url);
```

However, in this case, the database URL includes the username and password and has the following general form –

```
jdbc:oracle:driver:username/password@database
```

So, the above connection can be created as follows –

```
String URL = "jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);
```

Using a Database URL and a Properties Object

A third form of the DriverManager.getConnection( ) method requires a database URL and a Properties object –

```
DriverManager.getConnection(String url, Properties info);
```

A Properties object holds a set of keyword-value pairs. It is used to pass driver properties to the driver during a call to the getConnection() method.

To make the same connection made by the previous examples, use the following code –

```
import java.util.*;
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
Properties info = new Properties( );
info.put( "user", "username" );
info.put( "password", "password" );

Connection conn = DriverManager.getConnection(URL, info);
```

Closing JDBC Connections

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 could provide a 'finally' block in your code. A *finally* block always executes, regardless of an exception occurs or not.

To close the above opened connection, you should call close() method as follows –

```
conn.close();
```

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

Steps to Connect with DB

There are 5 steps to connect any java application with the database in java using JDBC

1. Load/Register the driver class
2. Create Connection Object
3. Create statement
4. Executing queries
5. Close connection

Loading the JDBC Driver

The first thing you need to do before you can open a database connection is to load the JDBC driver for the database

In this step of the jdbc connection process, we load the driver class by calling Class.forName() with the Driver class name as an argument. Once loaded, the Driver class creates an instance of itself. A client can connect to Database Server through JDBC Driver. Since most of the Database servers support ODBC driver therefore JDBC-ODBC Bridge driver is commonly used.

The return type of the Class.forName (String ClassName) method is “Class”. Class is a class in java.lang package.

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

Create Connection Object

The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager is considered the backbone of JDBC architecture. DriverManager class manages the JDBC drivers that are installed on the system.

The method **`DriverManager.getConnection(url, username, password)`** establishes a database connection. This method requires a database URL, which varies depending on your DBMS.

The getConnection() method is used to establish a connection to a database. It uses a username, password, and a **jdbc url** to establish a connection to the database and returns a connection object. A jdbc Connection represents a session/connection with a specific database. Within the context of a Connection, SQL, PL/SQL statements are executed and results are returned. An application can have one or more connections with a single database, or it can have many connections with different databases. A Connection object provides metadata i.e. information about the database, tables, and fields. It also contains methods to deal with transactions.

```
String dbURL = "jdbc:mysql://localhost:3306/sampledb";
String username = "root";
String password = "secret";

try {
     Connection conn = DriverManager.getConnection(dbURL, username, password);
     if (conn != null) {
        System.out.println("Connected");
    }
} catch (SQLException ex) {
    ex.printStackTrace();
}
```

Note:

- Typically, in the database URL, you also specify the name of an existing database to which you want to connect. For example, the URL `jdbc:mysql://localhost:3306/mysql` represents the database URL for the MySQL database named `mysql`. The samples in this tutorial use a URL that does not specify a specific database because the samples create a new database.

- 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`. The drivers for Java DB are `org.apache.derby.jdbc.EmbeddedDriver` and `org.apache.derby.jdbc.ClientDriver`, and the one for MySQL Connector/J is `com.mysql.jdbc.Driver`. See the documentation of your DBMS driver to obtain the name of the class that implements the interface `java.sql.Driver`.

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`.)

The method returns a `Connection` object, which represents a connection with the DBMS or a specific database. Query the database through this object.

Creating a jdbc Statement object

Once a connection is obtained we can interact with the database. Connection interface defines methods for interacting with the database via the established connection. To execute SQL statements, you need to instantiate a Statement object from your connection object by using the createStatement() method.

```
Statement statement = dbConnection.createStatement();
```

A statement object is used to send and execute SQL statements to a database.

Three kinds of Statements

1. **Statement**: Execute simple sql queries without parameters. The **createStatement**() creates an SQL Statement object and returns "Statement" Object
2. **PreparedStatement**: Execute precompiled sql queries with or without parameters. The prepareStatement(String sql) and returns "PreparedStatement" Object. PreparedStatement objects represent precompiled SQL statements.
3. **Callable** Statement: Execute a call to a database stored procedure. The prepareCall(String sql) returns a new CallableStatement object. CallableStatement objects are SQL stored procedure call statements.

Executing a SQL statement with the Statement object, and returning a jdbc resultSet.

Statement interface defines methods that are used to interact with database via the execution of SQL statements. The Statement class has three methods for executing statements:

1. **`public int executeUpdate(String query)`**
   Executes an INSERT, UPDATE or DELETE statement and returns an update account indicating number of rows affected (e.g. 1 row inserted, or 2 rows updated, or 0 rows affected).

2. **`public ResultSet executeQuery(String query)`**
   Executes a SELECT statement and returns a `ResultSet` object which contains results returned by the query.

3. **`public Boolean execute(String query)`**
   executes a general SQL statement. It returns true if the query returns a ResultSet, false if the query returns an update count or returns nothing. This method can be used with a Statement only.

Note:

- Statements that create a table, alter a table, or drop a table are all examples of DDL statements and are executed with the method executeUpdate().
- execute() method executes an SQL statement that is written as String object.

ResultSet

ResultSet provides access to a table of data generated by executing a Statement. The table rows are retrieved in sequence. A ResultSet maintains a cursor pointing to its current row of data. The next() method is used to successively step through the rows of the tabular results.

ResultSet contains table data returned by a SELECT query. Use this object to iterate over rows in the result set using **next**() method, and get value of a column in the current row using **getXXX()** methods (e.g. **getString()**, **getInt()**, **getFloat()** and so on). The column value can be retrieved either by index number (1-based) or by column name.

ResultSetMetaData Interface holds information on the types and properties of the columns in a ResultSet. It is constructed from the Connection object.

Microsoft Access

Note: The JDBC-ODBC Bridge driver (`sun.jdbc.odbc.JdbcOdbcDriver`) was removed from JDK 8! You need to use JDK 7 or find an alternate JDBC driver?!

MS Access is a personal database and is not meant for business production (it is probably sufficient for a small business). Microsoft has its own standard called ODBC, and does not provide a native JDBC drive for Access. A JDBC-ODBC bridge driver provided by Sun (called `sun.jdbc.odbc.JdbcOdbcDriver`) is used for JDBC programming. Some of the Java's JDBC features do not work on Access - due of the the limitations in the JDBC-ODBC bridge driver.

Access has various versions, such as Access 2010, Access 2007 and Access 2003, which are NOT 100% compatible. The file type for Access 2003 (and earlier versions) is "`.mdb`". The file type for Access 2007 is "`.accdb`".

Create Database in Access 2007

1. Launch "MS Access 2007".
2. Create a new database: From the "Access" button ⇒ "New" ⇒ In "Filename" box, select your working directory and enter "`ebookshop.accdb`" as the name of the database ⇒ "Create".
3. Delete the auto-generated column ID.
4. Click "Add New Field" to create the these 5 columns: "`id`", "`title`", "`author`", "`price`" and "`qty`".
5. Type in the above records.

ODBC (Open Database Connectivity) Connection

Close the Access before proceeding to next step to define ODBC connection (otherwise, you will get an error "invalid directory or file path", as the database is currently opened in an exclusive mode).

A so-called *ODBC* connection is needed to connect to a Microsoft database.

Define an ODBC connection called "`ebookshopODBC`", which selects the database we have just created, (i.e., "`ebookshop.mdb`" for Access 2003 or "`ebookshop.accdb`" for Access 2007), as follows:

1. Goto "Control Panel" ⇒ Administrator Tools ⇒ Data Source (ODBC),
2. Choose tab "System DSN" (System Data Source Name) (for all users in the system); or "User DSN" (User Data Source Name) (for the current login user only).
3. "Add" ⇒ Select "Microsoft Access Driver (*.mdb, *.accdb)" (for Access 2007); or "Microsoft Access Driver (*.mdb)" (for Access 2003) ⇒ Click "Finish".
4. In "ODBC Microsoft Access Setup" dialog: Enter "ebookshopODBC" in "Data Source Name", and use the "Select" button to navigate and select "ebookshop.accdb" (for Access 2007) or "ebookshop.mdb" (for Access 2003).

Database-URL

The `database-url` for Access is in the form of "`jdbc:odbc:{odbc-name}`", with protocol `jdbc` and sub-protocol `odbc`.

// Syntax

```
Connection conn = DriverManager.getConnection("jdbc:odbc:{odbc-name}");
```

// Example

```
Connection conn = DriverManager.getConnection("jdbc:odbc:ebookshopODBC");
```

For JDK prior to 1.6, you need to explicitly load the database drive as follows:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  // for MS Access/Excel
```

JDBC Program for MS Access

```
import java.sql.*;

public class DBDemo1
{
        public static void main(String[] args)
        {
                Connection con=null;

                try
                {
                        //1. Load/Register the Driver Class
                        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");


                        //2. Create/Establish the Connection
                        con = 
DriverManager.getConnection("jdbc:odbc:mydsn1","","");
```

```
                    //3. create the Statement Object
                    Statement st = con.createStatement();

                    //4. execute the queries/sql statements
                    String insertQuery = "insert into Employee values(102,
'Tanisha', 11000.00)";

                    int i = st.executeUpdate(insertQuery);

                    if(i>0)
                          System.out.println("\n\t Record is successfully
inserted");
                    else
                          System.out.println("\n\t Record not inserted");
             }
             catch(ClassNotFoundException ex)
             {
                    ex.printStackTrace();
             }
             catch(SQLException ex)
             {
                    ex.printStackTrace();
             }
             finally
             {
                    try
                    {
                          if(con!=null)
                          {
                                 //5. close the connection
                                 con.close();
                          }
                    }catch(SQLException ex){}
             }
      }
}
```

Update Records

```
String updateQuery = "update Employee set city='Mumbai' where EmpId=104";

int i = st.executeUpdate(updateQuery);

if(i>0)
      System.out.println("\n\t " + i  + " Records updated");
else
      System.out.println("\n\t Record not updated");
```

Delete Records

```
String deleteQuery = "DELETE from Employee where EmpID=104";

             int i = st.executeUpdate(deleteQuery );

             if(i>0)
                    System.out.println("\n\t " + i  + " Records Deleted");
             else
                    System.out.println("\n\t Record not Deleted");
```

Insert Record in Oracle Database

```
//1. Load/Register the Driver Class
Class.forName("oracle.jdbc.driver.OracleDriver");

//2. Create/Establish the Connection
con = 
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","ma
nager");

Statement st = con.createStatement();

String insertQuery = "insert into Employee values(102, 'Tanisha', 11000.00)";

int i = st.executeUpdate(insertQuery);

if(i>0)
       System.out.println("\n\t Record is successfully inserted");
else
       System.out.println("\n\t Record not inserted");
```

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

public class StatementDemo1 {
Connection con;
Statement stmt;
public StatementDemo1()
{
   try {
   Class.forName("com.mysql.jdbc.Driver");
   con = DriverManager.getConnection
   ("jdbc:mysql://localhost/test?user=root&password=root");
} catch (Exception e){
e.printStackTrace();
}
}

public String addCustomer(String custId, String name, String address,
String contact) {
String status = "", sql = "insert into Customer(custId,custName)
values('" + custId + "','" + name + "')";
if (address.trim().length() != 0) {
sql = "insert into Customer(custId,custName,custAddress)
values('" + custId + "','" + name + "','" + address + "')";
}
if (contact.trim().length() != 0) {
 sql = "insert into Customer(custId,custName,custAddress,custContact)
values('" + custId + "','" + name + "','" + address + "','" + contact + "')";
}
try {
    stmt = con.createStatement();
    int i = stmt.executeUpdate(sql);
    if (i != 0) {
        status = "Inserted";
    } else {
        status = "Not Inserted";
    }
} catch (Exception e) {
    e.printStackTrace();
}
return status;
}

public String editCustomer(String custid, String address, String contact) {
String status = "", sql = "";
if (address.trim().length() != 0) {
    sql = "update Customer set custAddress='" + address + "'
where custId='" + custid + "'";
}
if (contact.trim().length() != 0) {
    sql = "update Customer set custcontact='" + contact + "'
where custId='" + custid + "'";
}
if ((contact.trim().length() != 0) && (address.trim().length() != 0)) {
    sql = "update Customer set custAddress='" + address + "',
custcontact='" + contact + "' where custId='" + custid + "'";
}
if (sql.trim().length() == 0) {
    status = "Please provide new values";
} else {
    try {
        stmt = con.createStatement();
        int i = stmt.executeUpdate(sql);
        if (i != 0) {
            status = "Customer details updated successfully";
        } else {
            status = "Customer details not updated ";
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
return status;
}

public void searchCustomer(String custid) {
String sql = "";
if (custid.trim().length() == 0) {
    sql = "select * from Customer";
} else {
    sql = "select * from Customer where custid='" + custid + "'";
}
try {
    stmt = con.createStatement();
    ResultSet res = stmt.executeQuery(sql);
    while (res.next()) {
        System.out.print(res.getString(1));
        System.out.print("\t" + res.getString(2));
        System.out.print("\t" + res.getString(3));
        System.out.println("\t" + res.getString(4));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
}
```

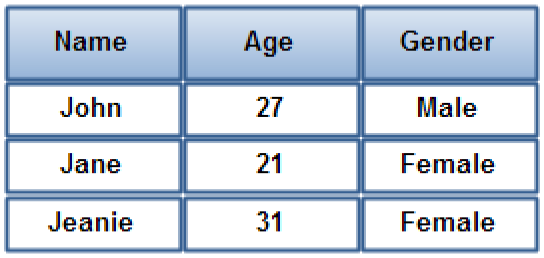
```
public String deleteCustomer(String custId) {
String status = "";
String sql = "delete from Customer where custid='" + custId + "'";
try {
    stmt = con.createStatement();
    int i = stmt.executeUpdate(sql);
    if (i != 0) {
        status = "Customer details deleted";
    } else {
        status = "Customer details not deleted";
    }
} catch (Exception e) {
    e.printStackTrace();
}
return status;
}

public void menuDisplay() {
String custId = "", custName = "", custAddress = "", custContact = "";
try {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    int ch = 0;
    while (true) {
    System.out.println("=== Customer Management System ======= \n"
          + "1. Add Customer \n "
                + "2. Edit Customer details \n "
                + "3. Delete Customer \n "
                + "4. Display Customer's record \n "
                + "5. Exit \n"
                + "Enter Choice \n");
        String str1 = br.readLine().toString();
        ch = Integer.parseInt(str1);
        switch (ch) {
        case 1: {
         // Customer Id can be left blank
         do {
         System.out.println("Enter Customer Id [ It can not be left blank ]");
         custId = br.readLine();
          } while (custId.trim().length() == 0);

         // Customer name can be left blank
          do {
            System.out.println("Enter Customer Name");
            custName = br.readLine();
             } while (custName.trim().length() == 0);

         System.out.println("Enter  Customer Address or Enter ]");
         custAddress = br.readLine();
         System.out.println("Enter Customer Contact No. or Enter ]");
         custContact = br.readLine();
         System.out.println(addCustomer(custId, custName, custAddress, custCon
tact));
         break;
            }
            case 2: {
           System.out.println("Customer address and contact no.");
                do {
                    System.out.println("Enter Customer Id");
                    custId = br.readLine();
                } while (custId.trim().length() == 0);

                System.out.println("Enter New Address or Enter ]");
                custAddress = br.readLine();
                System.out.println("Enter New Contact No. or Enter ]");
                custContact = br.readLine();
                System.out.println(editCustomer(custId, custAddress, custConta
ct));
                break;
            }
            case 3: {
                do {
                    System.out.println("Enter Customer Id to delete");
                    custId = br.readLine();
                } while (custId.trim().length() == 0);
                System.out.println(deleteCustomer(custId));
                break;
            }
            case 4: {
                System.out.println("Enter Customer Id or Enter]");
                custId = br.readLine();
                searchCustomer(custId);
                break;
            }
            case 5: {
                System.out.println(0);
            }
            default:
                break;
        }
    }
} catch (Exception e) {
    e.printStackTrace();
}
}

public static void main(String[] args) {
```

```
StatementDemo1 obj = new StatementDemo1();
obj.menuDisplay();
}
}
```

ResultSet

java.sql.ResultSet is an interface and it is used to retrieve SQL select query results. 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. It provides getXXX() methods to get data from each iteration. Here XXX represents datatypes.

A `ResultSet` Object consists of reco1rds. Each records contains a set of columns. Each record contains the same number of columns, although not all columns may have a value. A column can have a `null` value. Here is an illustration of a `ResultSet`:



Name	Age	Gender
John	27	Male
Jane	21	Female
Jeanie	31	Female

This `ResultSet` has 3 different columns (Name, Age, Gender), and 3 records with different values for each column.

Creating a ResultSet

You create a `ResultSet` by executing a `Statement` or `PreparedStatement`, like this:

```
Statement statement = connection.createStatement();

ResultSet result = statement.executeQuery("select * from people");
```

Or

```
String sql = "select * from people";

PreparedStatement statement = connection.prepareStatement(sql);

ResultSet result = statement.executeQuery();
```

Iterating the ResultSet

To iterate the `ResultSet` you use its `next()` method. The `next()` method returns true if the `ResultSet` has a next record, and moves the `ResultSet` to point to the next record. If there were no more records, `next()`returns false, and you can no longer. Once the `next()` method has returned false, you should not call it anymore. Doing so may result in an exception.

Here is an example of iterating a `ResultSet` using the `next()` method:

```
while(result.next()) {
    // ... get column values from this record
}
```

As you can see, the `next()` method is actually called before the first record is accessed. That means, that the `ResultSet` starts out pointing before the first record. Once `next()` has been called once, it points at the first record.

Similarly, when `next()` is called and returns false, the `ResultSet` is actually pointing after the last record.

You cannot obtain the number of rows in a `ResultSet` except if you iterate all the way through it and count the rows. However, if the `ResultSet` is forward-only, you cannot afterwards move backwards through it. Even if you could move backwards, it would a slow way of counting the rows in the `ResultSet.` You are better off structuring your code so that you do not need to know the number of records ahead of time.

Accessing Column Values

When iterating the `ResultSet` you want to access the column values of each record. You do so by calling one or more of the many `getXXX()` methods. You pass the name of the column to get the value of, to the many `getXXX()` methods. For instance:

```
while(result.next()) {
    result.getString    ("name");
    result.getInt       ("age");
    result.getBigDecimal("coefficient");

    // etc.
}
```

There are a lot of `getXXX()` methods you can call, which return the value of the column as a certain data type, e.g. String, int, long, double, BigDecimal etc. They all take the name of the column to obtain the column value for, as parameter

The getXXX() methods also come in versions that take a column index instead of a column name. For instance:

```
while(result.next()) {
    result.getString    (1);
    result.getInt       (2);
    result.getBigDecimal(3);

    // etc.
}
```

The index of a column typically depends on the index of the column in the SQL statement. For instance, the SQL statement

```
select name, age, coefficient from person
```

This statement returns data which has three columns. The column name is listed first, and will thus have index 1 in the ResultSet. The column age will have index 2, and the column coefficient will have index 3.
Sometimes you do not know the index of a certain column ahead of time. For instance, if you use a `select * from` type of SQL query, you do not know the sequence of the columns.

If you do not know the index of a certain column you can find the index of that column using the `ResultSet.findColumn(String columnName)` method, like this:

```
int nameIndex   = result.findColumn("name");
int ageIndex    = result.findColumn("age");
int coeffIndex  = result.findColumn("coefficient");

while(result.next()) {
    String name = result.getString(nameIndex);
    int age = result.getInt(ageIndex);
    BigDecimal coefficient = result.getBigDecimal(coeffIndex);
}
```

Fetch All data from Database and Display

```
import java.sql.*;

public class DBDemo2
{
      public static void main(String[] args)
      {
            Connection con=null;

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

                  String sql= "SELECT * from Employee";

                  ResultSet rs = st.executeQuery(sql);

                  while(rs.next())
                  {
                        int id = rs.getInt(1);
                        String ename = rs.getString(2);
                        String city = rs.getString(3);
                        double salary = rs.getDouble(4);

                        System.out.println("\n\t " + id + "\t"+ ename 
+"\t" + city + "\t" + salary);
                  }

            }
            catch(ClassNotFoundException ex)
            {
                  ex.printStackTrace();
            }
            catch(SQLException ex)
            {
                  ex.printStackTrace();
            }
            finally
            {
                  try
                  {
                        if(con!=null)
                        {
                              //5. close the connection
                              con.close();
                        }
                  }catch(SQLException ex){}
            }
      }
}
```

Getting Specific Records from Database

String sql= "SELECT EName,Salary, City from Employee";

```
ResultSet rs = st.executeQuery(sql);

int indEname = rs.findColumn("EName");
int indSalary = rs.findColumn("Salary");

while(rs.next())
{

      String ename = rs.getString(indEname);
      double salary = rs.getDouble(indSalary);

      System.out.println("\n\t " + ename + "   " + salary);
}
```

Using generic execute() method to submit any type of queries to database

The execute() can executes any SQL statement, which may return multiple results.

Normally you can ignore this multiple results future unless you are executing a stored procedure that you know may return multiple results.

The Statement interface has these methods for that:

Methods in Statement interface	Description
boolean execute(String sql)	Executes the given SQL statement, and returns true if the first result is a ResultSet object; false if it is an update count or there are no results.
boolean execute(String sql, int autoGeneratedKeys)	Same as execute(String sql) but for a SQL INSERT you can signals the driver a given flag about whether the auto-generated keys produced by this Statement object should be made available for retrieval.
boolean execute(String sql, int[] columnIndexes)	Same as execute(String sql) but for a SQL INSERT you can signals the driver that the auto-generated keys indicated in the given array should be made available for retrieval. The array contains the indexes (starting with 1 for the first) of the columns.
boolean execute(String sql, String[] columnNames)	Same as execute(String sql) but for a SQL INSERT you can signals the driver that the auto-generated keys indicated in the given

	array should be made available for retrieval. The array contains the names of the columns.

As SQL statements used with an execute() method can lead to a ResultSet result, then you may be need to read the ResultSet data.

The three last methods above can result in a possibility to retrieve the auto generated key for a INSERT SQL.

The Statement interface has some methods that help you to retrieve such things after an SQL execute() method:

Methods in Statement interface	Description
ResultSet getResultSet()	Retrieves the current result as a ResultSet object. This method should be called only once per result.
int getUpdateCount()	Retrieves the current result as an update count; if the result is a ResultSet object or there are no more results, -1 is returned. This method should be called only once per result.
boolean getMoreResults()	Moves to this Statement object's next result, returns true if it is a ResultSet object, and implicitly closes any current ResultSet object(s) obtained with the method getResultSet.
ResultSet getGeneratedKeys()	Retrieves any auto-generated keys created as a result of executing this Statement object. If this Statement object did not generate any keys, an empty ResultSet object is returned.

To test for multiple ResultSet you can use this:

```
// There are no more results when the following is true:
// stmt is a Statement object
       ((stmt.getMoreResults() == false) && (stmt.getUpdateCount() == -1))
```

SQL Update with the execute() method contained in a JDBC-Statement object.

1. First you need to Create a Database Connection.
2. Then create a regular Statement object with the method, createStatement().
3. The next step is to write your SQL Update statement and give this string as a parameter to the method:

Method in Statement interface	Description
boolean execute(String sql)	Executes the given SQL statement, and returns true if the first result is a ResultSet object; false if it is an update count or there are no results.

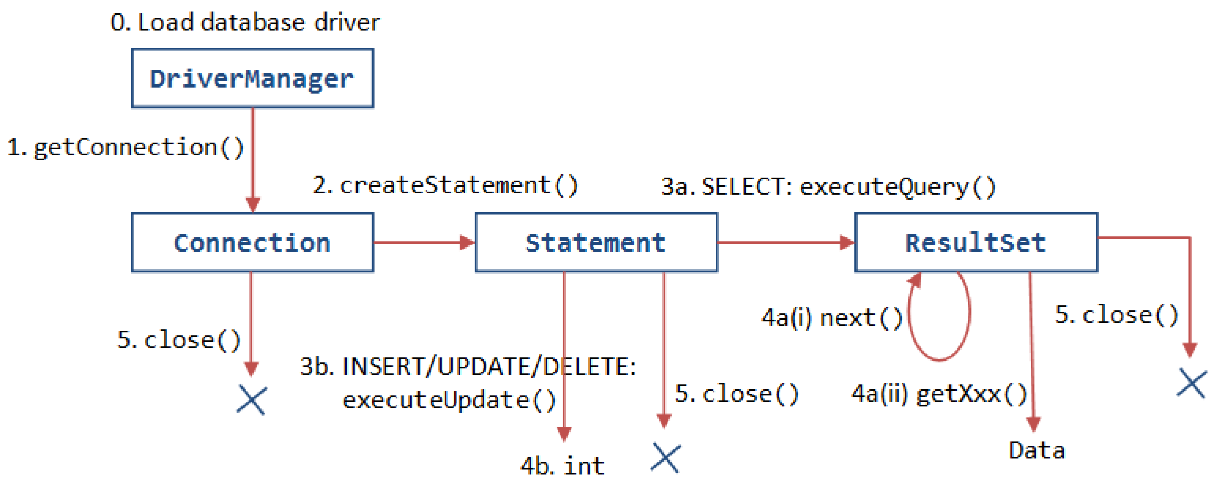
Example of using this method to Update a database table:

SQL Delete or Insert with Statement object execute()method

1. First you need to Create a Database Connection.
2. When you have got the database connection, create a regular Statement object with the method,createStatement().
3. The next step is to write your SQL Delete, SQL Insert or SQL Select statement and give this string as a parameter to the method:

Method in Statement interface	Description
boolean execute(String sql)	Executes the given SQL statement, and returns true if the first result is a ResultSet object; false if it is an update count or there are no results.

JDBC Statement interface



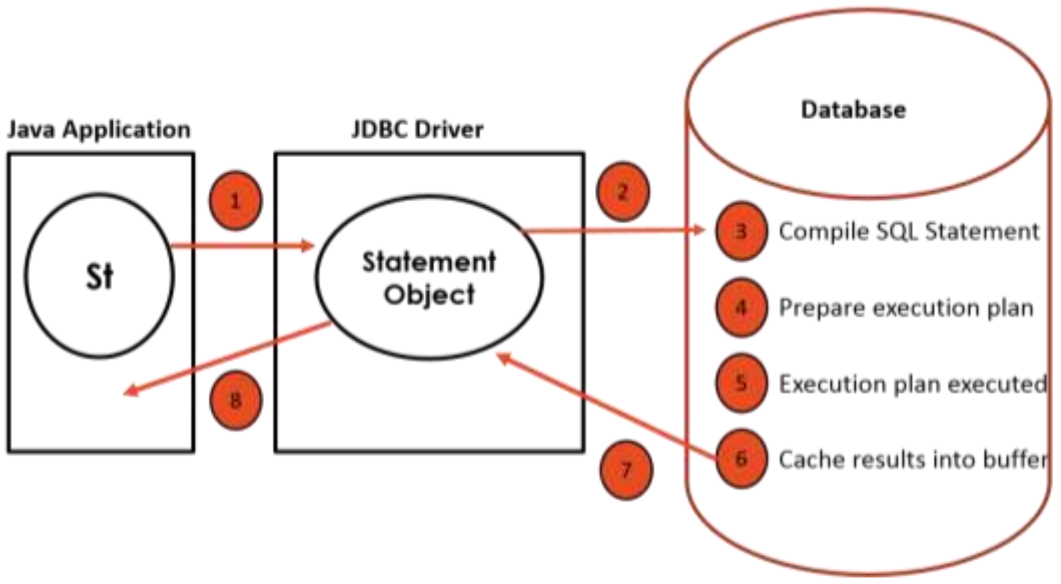
The java.sql.Statement interface object represent static SQL statement. It is base object type of SQL statement in java. Only one ResultSet object is associated with one Statement object. The PreparedStatement interface is the derived from it.

To use Statement interface in source code, first we need to create object of Statement by calling createStatement() method from java.sql.Connection interface. The createStatement() is available in java.sql.Connection interface. It does not take any argument.

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

After creating object of Statement type object, we call **executeUpdate()** or **executeQuery()** method. It depends whether database is updating or getting result by SQL statement. If database is updating

(Insert, Update, Delete or Create) call **int executeUpdate()** and record is fetching call **ResultSet executeQuery()**.



Process flow of Statement Object

- The executeXXX() method the Statement object submits the SQL statement to database.
- The Statement object submits the SQL Statements to database
- The database compiles the given SQL statement (at database side)
- An execution plan is prepared by database to execute statements
- The execution plan for the compiled SQL statement is executed
- If the SQL statement requires data retrieval (select), then the database simply caches the results of SQL statements into buffer
- Finally, the response is sent to Java

Problems with Statement Object

- Using Statement Object, compilation of query is costly i.e. takes more time in syntax checking, name validation, and pseudo code generation after the query
- The query optimizer evaluates different plan for executing query nd then it returns the results
- If you are passing same SQL statements it compiles again and again and it is a time consuming process

PreparedStatement

A `PreparedStatement` is a special kind of `Statement` object with some useful features. Remember, you need a `Statement` in order to execute either a query or an update. The PreparedStatement interface extends the Statement interface which gives you added functionality with a couple of advantages over a generic Statement object

- It is used to execute precompiled query, which can be executed multiple times without compiling again and again
- It improves the performance of application
- It uses setter and getter methods for updating SQL 99 types
- It represents and executes only one SQL statement

Creating a PreparedStatement

To execute any precompiled statement using PreparedStatement we must follow below steps

1. Create a PreparedStatement object
2. Supply the values of preparedStatment parameters
3. Execute the SQL Statement

Before you can use a `PreparedStatement` you must first create it. You do so using the `Connection.prepareStatement()`, like this:

```
String sql = "select * from people where id=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);
```

The `PreparedStatement` is now ready to have parameters inserted.

Supply values to Parameters of PreparedStatement

Everywhere you need to insert a parameter into your SQL, you write a question mark (?). For instance:

```
String sql = "select * from people where id=?";
```

Once a `PreparedStatement` is created (prepared) for the above SQL statement, you can insert parameters at the location of the question mark. This is done using the many `setXXX()` methods. Here is an example:

```
preparedStatement.setLong(1, 123);
```

The first number (1) is the index of the parameter to insert the value for. The second number (123) is the value to insert into the SQL statement.

Here is the same example with a bit more details:

```
String sql = "select * from people where id=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);
```

```
    preparedStatement.setLong(123);
```

You can have more than one parameter in an SQL statement. Just insert more than one question mark. Here is a simple example:

```
    String sql = "select * from people where firstname=? and lastname=?";

    PreparedStatement preparedStatement =
            connection.prepareStatement(sql);

    preparedStatement.setString(1, "John");
    preparedStatement.setString(2, "Smith");
```

Executing the PreparedStatement

Executing the `PreparedStatement` looks like executing a regular `Statement`. To execute a query, call the `executeQuery()` or `executeUpdate` method. Here is an `executeQuery()` example:

```
    String sql = "select * from people where firstname=? and lastname=?";

    PreparedStatement preparedStatement =
            connection.prepareStatement(sql);

    preparedStatement.setString(1, "John");
    preparedStatement.setString(2, "Smith");

    ResultSet result = preparedStatement.executeQuery();
```

As you can see, the `executeQuery()` method returns a ResultSet. Iterating the ResultSet is described in the Query the Database text.

Here is an executeUpdate() example:

```
    String sql = "update people set firstname=? , lastname=? where id=?";

    PreparedStatement preparedStatement =
            connection.prepareStatement(sql);

    preparedStatement.setString(1, "Gary");
    preparedStatement.setString(2, "Larson");
    preparedStatement.setLong  (3, 123);

    int rowsAffected = preparedStatement.executeUpdate();
```

The `executeUpdate()` method is used when updating the database. It returns an int which tells how many records in the database were affected by the update.

Reusing a PreparedStatement

Once a PreparedStatement is prepared, it can be reused after execution. You reuse a PreparedStatementby setting new values for the parameters and then execute it again. Here is a simple example:

```
String sql = "update people set firstname=? , lastname=? where id=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);

preparedStatement.setString(1, "Gary");
preparedStatement.setString(2, "Larson");
preparedStatement.setLong  (3, 123);

int rowsAffected = preparedStatement.executeUpdate();

preparedStatement.setString(1, "Stan");
preparedStatement.setString(2, "Lee");
preparedStatement.setLong  (3, 456);

int rowsAffected = preparedStatement.executeUpdate();
```

This works for executing queries too, using the `executeQuery()` method, which returns a `ResultSet`.

PreparedStatement Performance

It takes time for a database to parse an SQL string, and create a query plan for it. A query plan is an analysis of how the database can execute the query in the most efficient way.

If you submit a new, full SQL statement for every query or update to the database, the database has to parse the SQL and for queries create a query plan. By reusing an existing `PreparedStatement` you can reuse both the SQL parsing and query plan for subsequent queries. This speeds up query execution, by decreasing the parsing and query planning overhead of each execution.

There are two levels of potential reuse for a `PreparedStatement`.

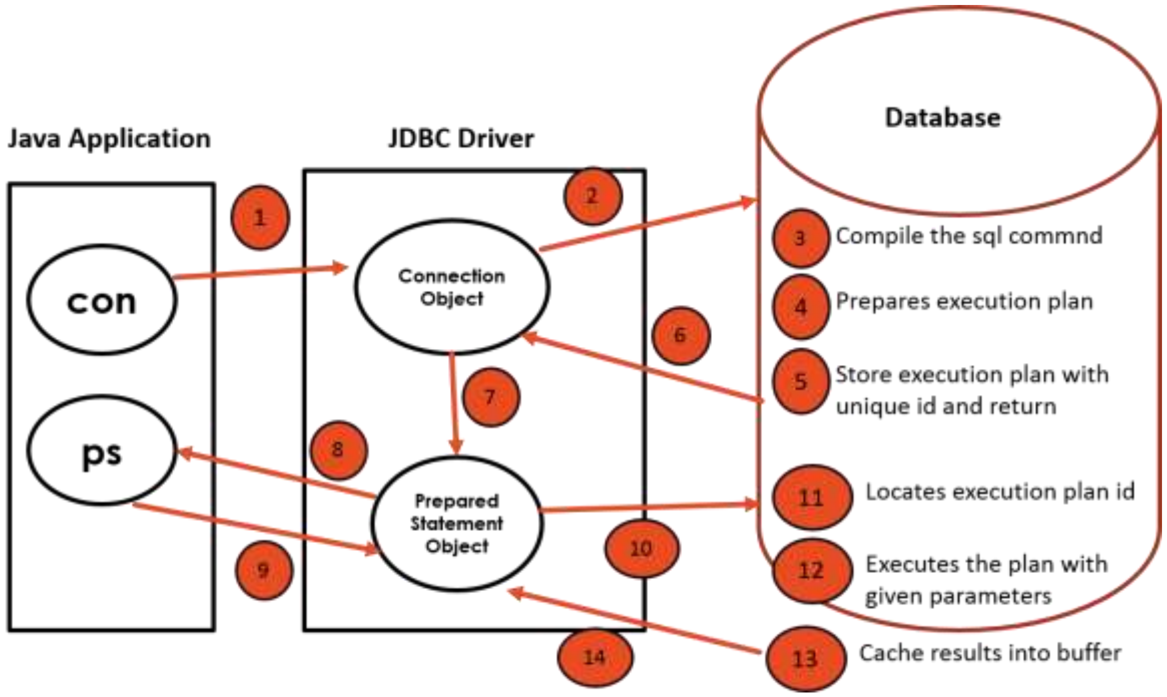
1. Reuse of PreparedStatement by the JDBC driver.
2. Reuse of PreparedStatement by the database.

First of all, the JDBC driver can cache `PreparedStatement` objects internally, and thus reuse the `PreparedStatement` objects. This may save a little of the `PreparedStatement` creation time.

Second, the cached parsing and query plan could potentially be reused across Java applications, for instance application servers in a cluster, using the same database.

Execution process of PreparedStatment

- The prepareStatement() of connection interface is used to get the PreparedStatement Object
- Connection object submits the given SQL command to database
- The database compiles the SQL commands
- Execution plan is prepared by database to execute commands
- The database stores the execution plan with a unique ID and returns the same to connection object
- Connection object prepares a PreparedStatement Object and initializes with execution plan ID
- The setXXX() methods of PreparedStatment object are used to set the parameters to the SQL command placeholders (?)
- The executexxx() methods of PreparedStatement object is invoked to execute the SQL command with the parameters
- The PreparedStatement object delegates the request to database
- The Database locates and executes exection plan ID with given parameters


Java Application
JDBC Driver
Database
con
ps
Connection Object
Prepared Statement Object
1
2
3 Compile the sql commnd
4 Prepares execution plan
5 Store execution plan with unique id and return
6
7
8
9
10
11 Locates execution plan id
12 Executes the plan with given parameters
13 Cache results into buffer
14

- Finally the result of the SQL

Difference between PreparedStatement and Statement

Statement	PreparedStatement

1	Statement interface is slow because it compiles the queries for each execution	PreparedStatement interface is faster, because its compile the command for once.
2	We cannot use ? symbol in sql command so setting dynamic value into the command is complex	We can use ? symbol in sql command, so setting dynamic value is simple.
3	We cannot use statement for writing or reading binary data (picture)	We can use PreparedStatement for reading or writing binary data.

```
import java.sql.*;

public class PreparedStatementDemo
{
	public static void main(String[] args)throws ClassNotFoundException, 
SQLException
	{
		Class.forName("oracle.jdbc.driver.OracleDriver");
		Connection con = 
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","m
anager");

		String sql = "INSERT INTO Employee values(?,?,?,?)";
		PreparedStatement ps = con.prepareStatement(sql);

		ps.setInt(1, 10);
		ps.setString(2, "Sonali");
		ps.setString(3, "Sangli");
		ps.setDouble(4, 10000.00);

		int i = ps.executeUpdate();
		if(i>0)
			System.out.println("\n\t " + i + " records inserted");
		else
			System.out.println("\n\t Record not inserted");

		con.close();
	}
}
```

Update Example

```
	String sql = "UPDATE Employee set City=? where EmpId=?";
	PreparedStatement ps = con.prepareStatement(sql);
```

```
ps.setString(1, "Mumbai");
ps.setInt(2, 2);

int i = ps.executeUpdate();
```

Delete Example

```
String sql = "Delete From Employee where EmpId=?";
PreparedStatement ps = con.prepareStatement(sql);

ps.setInt(1, 7);

int i = ps.executeUpdate();
```

Select Example

```
String sql = "Select * from Employee";
PreparedStatement ps = con.prepareStatement(sql);
ResultSet rs = ps.executeQuery()

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

ResultSet Types

A `ResultSet` can be of a certain type. The type determines some characteristics and abilities of the`ResultSet`.

Not all types are supported by all databases and JDBC drivers. You will have to check your database and JDBC driver to see if it supports the type you want to use. The `DatabaseMetaData.supportsResultSetType(int type)` method returns true or false depending on whether the given type is supported or not. The `DatabaseMetaData` class is covered in a later text.

At the time of writing there are three `ResultSet` types:

1. ResultSet.TYPE_FORWARD_ONLY
2. ResultSet.TYPE_SCROLL_INSENSITIVE
3. ResultSet.TYPE_SCROLL_SENSITIVE

The default type is `TYPE_FORWARD_ONLY`

`TYPE_FORWARD_ONLY` means that the `ResultSet` can only be navigated forward. That is, you can only move from row 1, to row 2, to row 3 etc. You cannot move backwards in the `ResultSet`.

`TYPE_SCROLL_INSENSITIVE` means that the `ResultSet` can be navigated (scrolled) both forward and backwards. You can also jump to a position relative to the current position, or jump to an absolute position. The `ResultSet` is insensitive to changes in the underlying data source while the `ResultSet` is open. That is, if a record in the `ResultSet` is changed in the database by another thread or process, it will not be reflected in already opened `ResulsSet`'s of this type.

`TYPE_SCROLL_SENSITIVE` means that the `ResultSet` can be navigated (scrolled) both forward and backwards. You can also jump to a position relative to the current position, or jump to an absolute position. The `ResultSet` is sensitive to changes in the underlying data source while the `ResultSet` is open. That is, if a record in the `ResultSet` is changed in the database by another thread or process, it will be reflected in already opened `ResulsSet`'s of this type.

Navigation Methods

The `ResultSet` interface contains the following navigation methods. Remember, not all methods work with all `ResultSet` types. What methods works depends on your database, JDBC driver, and the `ResultSet`type.

Method	Description
`absolute()`	Moves the `ResultSet` to point at an absolute position. The position is a row number passed as parameter to the `absolute()` method.
`afterLast()`	Moves the `ResultSet` to point after the last row in the `ResultSet`.
`beforeFirst()`	Moves the `ResultSet` to point before the first row in the `ResultSet`.
`first()`	Moves the `ResultSet` to point at the first row in the `ResultSet`.
`last()`	Moves the `ResultSet` to point at the last row in the `ResultSet`.
`next()`	Moves the `ResultSet` to point at the next row in the `ResultSet`.
`previous()`	Moves the `ResultSet` to point at the previous row in the `ResultSet`.
`relative()`	Moves the `ResultSet` to point to a position relative to its current position. The relative position is passed as a parameter to the relative method, and can be both positive and negative.
	Moves the `ResultSet`

The `ResultSet` interface also contains a set of methods you can use to inquire about the current position of the `ResultSet`. These are:

Method	Description
`getRow()`	Returns the row number of the current row - the row currently pointed to by the`ResultSet`.
`getType()`	Returns the `ResultSet` type.
`isAfterLast()`	Returns true if the `ResultSet` points after the last row. False if not.
`isBeforeFirst()`	Returns true if the `ResultSet` points before the first row. False if not.
`isFirst()`	Returns true if the `ResultSet` points at the first row. False if not.

Finally the `ResultSet` interface also contains a method to update a row with database changes, if the`ResultSet` is sensitive to change.

Method	Description
`refreshRow()`	Refreshes the column values of that row with the latest values from the database.

ResultSet Concurrency

The `ResultSet` concurrency determines whether the `ResultSet` can be updated, or only read.

Some databases and JDBC drivers support that the `ResultSet` is updated, but not all databases and JDBC drivers do. The `DatabaseMetaData.supportsResultSetConcurrency(int concurrency)` method returns true or false depending on whether the given concurrency mode is supported or not. The `DatabaseMetaData` class is covered in a later text.

A `ResultSet` can have one of two concurrency levels:

1. ResultSet.CONCUR_READ_ONLY
2. ResultSet.CONCUR_UPDATABLE

CONCUR_READ_ONLY means that the `ResultSet` can only be read.

CONCUR_UPDATABLE means that the `ResultSet` can be both read and updated.

Updating a ResultSet

If a `ResultSet` is updatable, you can update the columns of each row in the `ResultSet`. You do so using the many `updateXXX()` methods. For instance:

```
    result.updateString     ("name"       , "Alex");
    result.updateInt        ("age"        , 55);
    result.updateBigDecimal ("coefficient", new BigDecimal("0.1323");
    result.updateRow();
```

You can also update a column using column index instead of column name. Here is an example:

```
    result.updateString     (1, "Alex");
    result.updateInt        (2, 55);
    result.updateBigDecimal (3, new BigDecimal("0.1323");
    result.updateRow();
```

Notice the updateRow() call. It is when updateRow() is called that the database is updated with the values of the row. Id you do not call this method, the values updated in the ResultSet are never sent to the database. If you call updateRow() inside a transaction, the data is not actually committed to the database until the transaction is committed.

Inserting Rows into a ResultSet

If the `ResultSet` is updatable it is also possible to insert rows into it. You do so by:

1. call ResultSet.moveToInsertRow()
2. update row column values
3. call ResultSet.insertRow()

Here is an example:

```
result.moveToInsertRow();
result.updateString     (1, "Alex");
result.updateInt        (2, 55);
result.updateBigDecimal (3, new BigDecimal("0.1323");
result.insertRow();

result.beforeFirst();
```

The row pointed to after calling `moveToInsertRow()` is a special row, a buffer, which you can use to build up the row until all column values has been set on the row.

Once the row is ready to be inserted into the `ResultSet`, call the `insertRow()` method.

After inserting the row the `ResultSet` still pointing to the insert row. However, you cannot be certain what will happen if you try to access it, once the row has been inserted. Therefore you should move the`ResultSet` to a valid position after inserting the new row. If you need to insert another row, explicitly call`moveToInsertRow()` to signal this to the `ResultSet`.

ResultSet Holdability

The `ResultSet` holdability determines if a `ResultSet` is closed when the `commit()` method of the underlying`connection` is called.

Not all holdability modes are supported by all databases and JDBC drivers.
The`DatabaseMetaData.supportsResultSetHoldability(int holdability)` returns true or false depending on whether the given holdability mode is supported or not.
The `DatabaseMetaData` class is covered in a later text.

There are two types of holdability:

1. ResultSet.CLOSE_CURSORS_OVER_COMMIT
2. ResultSet.HOLD_CURSORS_OVER_COMMIT

The `CLOSE_CURSORS_OVER_COMMIT` holdability means that all `ResultSet` instances are closed when`connection.commit()` method is called on the connection that created the `ResultSet`.

The `HOLD_CURSORS_OVER_COMMIT` holdability means that the `ResultSet` is kept open when the`connection.commit()` method is called on the connection that created the `ResultSet`.

The `HOLD_CURSORS_OVER_COMMIT` holdability might be useful if you use the `ResultSet` to update values in the database. Thus, you can open a `ResultSet`, update rows in it,
call `connection.commit()` and still keep the same `ResultSet` open for future transactions on the same rows.

Handling NULL Values

The Null values must be checked before they appear on the application. Since, Null values should not be displayed in real time applications, so they must be handled

To identify Null values, we can make use of ResultSet interface methods

public Boolean wasNull() method returns true if the obtained column contains Null values otherwise it returns false

SQL's use of NULL values and Java's use of null are different concepts. So, to handle SQL NULL values in Java, there are three tactics you can use –

- Avoid using getXXX( ) methods that return primitive data types.
- Use wrapper classes for primitive data types, and use the ResultSet object's wasNull( ) method to test whether the wrapper class variable that received the value returned by the getXXX( ) method should be set to null.
- Use primitive data types and the ResultSet object's wasNull( ) method to test whether the primitive variable that received the value returned by the getXXX( ) method should be set to an acceptable value that you've chosen to represent a NULL.

Here is one example to handle a NULL value –

```
Statement stmt = conn.createStatement( );
String sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);

int id = rs.getInt(1);
if( rs.wasNull( ) ) {
   id = 0;
}
```

Inserting Images into Database

You can store images in the database in java by the help of PreparedStatement interface

The setBinaryStream() method of PreparedStatement is used to set Binary information into the parameterIndex

The syntax of setBinaryStream() method is given below:

1) public void setBinaryStream(int paramIndex,InputStream stream)

throws SQLException

2) public void setBinaryStream(int paramIndex,InputStream stream,long length)

throws SQLException

For storing image into the database, BLOB (Binary Large Object) datatype is used in the table

BLOB is nothing bug Binary Large Object. BLOB is used to store large amount of binary data into database like images, etc. Below example shows how to store images into database rows.

```
public class MyBlobInsert {

     public static void main(String a[]){
        Connection con = null;
        PreparedStatement ps = null;
        InputStream is = null;
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con = DriverManager.
                    getConnection("jdbc:oracle:thin:@<hostname>:<port num>:<DB name>"
                        ,"user","password");
            ps = con.prepareCall("insert into student_profile values (?,?)");
            ps.setInt(1, 101);
            is = new FileInputStream(new File("Student_img.jpg"));
            ps.setBinaryStream(2, is);
            int count = ps.executeUpdate();
            System.out.println("Count: "+count);
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } finally{
            try{
                if(is != null) is.close();
                if(ps != null) ps.close();
                if(con != null) con.close();
            } catch(Exception ex){}
        }
    }
}
```

read an image from database table? or Write an example for reading BLOB from table.

```
public class MyBlobRead {


    public static void main(String a[]){


        Connection con = null;
        Statement st = null;
        ResultSet rs = null;
        InputStream is = null;
        OutputStream os = null;
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con = DriverManager.
                    getConnection("jdbc:oracle:thin:@<hostname>:<port num>:<DB name>
                        ,"user","password");
            st = con.createStatement();
            rs = st.executeQuery("select student_img from student_profile where
id=101");
            if(rs.next()){
                is = rs.getBinaryStream(1);
            }
            is = new FileInputStream(new File("Student_img.jpg"));
            os = new FileOutputStream("std_img.jpg");
            byte[] content = new byte[1024];
            int size = 0;
            while((size = is.read(content)) != -1){
                os.write(content, 0, size);
            }
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } finally{
            try{
                if(is != null) is.close();
                if(os != null) os.close();
                if(st != null) st.close();
                if(con != null) con.close();
            } catch(Exception ex){}
        }
    }
}
```

Insert file data into MySQL database using JDBC

To store a file into a database table, the table must have a column whose data type is BLOB (*Binary Large OBject*). Assuming we have a MySQL table called `person` which is created by the following SQL script:

```
CREATE TABLE `person` (
  `person_id` int(11) NOT NULL AUTO_INCREMENT,
  `first_name` varchar(45) DEFAULT NULL,
  `last_name` varchar(45) DEFAULT NULL,
  `photo` mediumblob,
  PRIMARY KEY (`person_id`)
)
```

We can notice that the column `photo` has type of `mediumblob` - which is one of four MySQL's blob types:

- TINYBLOB: 255 bytes
- BLOB: 65,535 bytes (64 KB)
- MEDIUMBLOB: 16,777,215 bytes (16 MB)
- LONGBLOB: 4 GB

That means the photo column can store a file up to 16 MB. You can choose which blob type

Insert file using standard JDBC API (database independent)

To store content of a file (binary data) into the table, we can use the following method defined by the interface

```
void setBlob(int parameterIndex, InputStream inputStream)
```

And we have to supply an input stream of the file to be stored. For example:

```
String filePath = "D:/Photos/Tom.jpg";
InputStream inputStream = new FileInputStream(new File(filePath));
String sql = "INSERT INTO person (photo) values (?)";
PreparedStatement statement = connection.prepareStatement(sql);
statement.setBlob(1, inputStream);
statement.executeUpdate();
```

Example

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
```

```
public class JdbcInsertFileOne {

    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/contactdb";
        String user = "root";
        String password = "secret";

        String filePath = "D:/Photos/Tom.png";

        try {
            Connection conn = DriverManager.getConnection(url, user, password);

            String sql = "INSERT INTO person (first_name, last_name, photo) values
(?, ?, ?)";
            PreparedStatement statement = conn.prepareStatement(sql);
            statement.setString(1, "Tom");
            statement.setString(2, "Eagar");
            InputStream inputStream = new FileInputStream(new File(filePath));

            statement.setBlob(3, inputStream);

            int row = statement.executeUpdate();
            if (row > 0) {
                System.out.println("A contact was inserted with photo image.");
            }
            conn.close();
        } catch (SQLException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Retrieve File Data from database

Using **PreparedStatement** we can retrieve and store the image in the database.

The **getBlob()** method of PreparedStatement is used to get Binary information, it returns the instance of Blob. After calling the**getBytes()** method on the blob object, we can get the array of binary information that can be written into the image file.

```
public  byte[] getBytes(long pos, int length)throws SQLException
```

File data is usually stored in database in column of BLOB type, so with JDBC we can use the method getBlob() defined in the java.sql.ResultSet interface. Like other getXXX() methods of the ResultSet interface, there are also two variants of the getBlob()method:

- `Blob getBlob(int columnIndex)`
- `Blob getBlob(String columnLabel)`

```
import java.sql.*;
import java.io.*;
public class RetrieveImage {
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 imgtable");
                ResultSet rs=ps.executeQuery();
                if(rs.next()){//now on 1st row
                        Blob b=rs.getBlob(2);//2 means 2nd column data
                        byte barr[]=b.getBytes(1,(int)b.length());//1 means first image

                        FileOutputStream fout=new FileOutputStream("d:\\sonoo.jpg");
                        fout.write(barr);

                        fout.close();
                }//end of if
        System.out.println("ok");
```

```
            con.close();
        }catch (Exception e) {e.printStackTrace();  }
    }
}
```

Both of these methods return a java.sql.Blob object from which we can obtain an InputStream in order to read the binary data. For example:

```
ResultSet result = statement.executeQuery();
if (result.next()) {
    Blob blob = result.getBlob("photo");
    InputStream inputStream = blob.getBinaryStream();
    // read the input stream...

}
```

The above code snippet retrieves blob data from the column photo, of the current result set. Then obtain the input stream by invoking the method getBinaryStream() on the Blob object. Reading the input stream, in conjunction with anOutputStream, we can save the binary data into a file, as shown in the following example program:

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.sql.Blob;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * This program demonstrates how to read file data from database and save the
 * data into a file on disk.
 * @author www.codejava.net
 *
 */
public class JdbcReadFile {
    private static final int BUFFER_SIZE = 4096;
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/contactdb";
        String user = "root";
        String password = "secret";

        String filePath = "D:/Photos/Tom.jpg";

        try {
            Connection conn = DriverManager.getConnection(url, user, password);

            String sql = "SELECT photo FROM person WHERE first_name=? AND
last_name=?";
            PreparedStatement statement = conn.prepareStatement(sql);
            statement.setString(1, "Tom");
            statement.setString(2, "Eagar");

            ResultSet result = statement.executeQuery();
            if (result.next()) {
                Blob blob = result.getBlob("photo");
                InputStream inputStream = blob.getBinaryStream();
                OutputStream outputStream = new FileOutputStream(filePath);

                int bytesRead = -1;
                byte[] buffer = new byte[BUFFER_SIZE];
                while ((bytesRead = inputStream.read(buffer)) != -1) {
                    outputStream.write(buffer, 0, bytesRead);
                }

                inputStream.close();
                outputStream.close();
                System.out.println("File saved");
            }
            conn.close();
        } catch (SQLException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

JDBC - Data Types

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

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

SQL	JDBC/Java	setXXX	updateXXX
VARCHAR	java.lang.String	setString	updateString
CHAR	java.lang.String	setString	updateString
LONGVARCHAR	java.lang.String	setString	updateString
BIT	boolean	setBoolean	updateBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	updateBigDecimal
TINYINT	byte	setByte	updateByte
SMALLINT	short	setShort	updateShort
INTEGER	int	setInt	updateInt
BIGINT	long	setLong	updateLong
REAL	float	setFloat	updateFloat
FLOAT	float	setFloat	updateFloat
DOUBLE	double	setDouble	updateDouble
VARBINARY	byte[ ]	setBytes	updateBytes
BINARY	byte[ ]	setBytes	updateBytes
DATE	java.sql.Date	setDate	updateDate
TIME	java.sql.Time	setTime	updateTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	updateTimestamp
CLOB	java.sql.Clob	setClob	updateClob
BLOB	java.sql.Blob	setBlob	updateBlob

ARRAY	java.sql.Array	setARRAY	updateARRAY
REF	java.sql.Ref	SetRef	updateRef
STRUCT	java.sql.Struct	SetStruct	updateStruct

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

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

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

SQL	JDBC/Java	setXXX	getXXX
VARCHAR	java.lang.String	setString	getString
CHAR	java.lang.String	setString	getString
LONGVARCHAR	java.lang.String	setString	getString
BIT	boolean	setBoolean	getBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	getBigDecimal
TINYINT	byte	setByte	getByte
SMALLINT	short	setShort	getShort
INTEGER	int	setInt	getInt
BIGINT	long	setLong	getLong
REAL	float	setFloat	getFloat
FLOAT	float	setFloat	getFloat
DOUBLE	double	setDouble	getDouble
VARBINARY	byte[ ]	setBytes	getBytes
BINARY	byte[ ]	setBytes	getBytes
DATE	java.sql.Date	setDate	getDate
TIME	java.sql.Time	setTime	getTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	getTimestamp
CLOB	java.sql.Clob	setClob	getClob
BLOB	java.sql.Blob	setBlob	getBlob
ARRAY	java.sql.Array	setARRAY	getARRAY
REF	java.sql.Ref	SetRef	getRef
STRUCT	java.sql.Struct	SetStruct	getStruct

Date & Time Data Types

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

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

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

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

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

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

Microsoft Excel

Surprisingly, you can use SQL commands to manipulate Excel spreadsheet. This is handy to export data into Excel, says from a database.

Setup Excel Spreadsheet

Create an Excel Spreadsheet called "`ebookshop.xlsx`" (corresponding to database name). Create a sheet called "`books`" (corresponding to table name). Create 5 column headers: id, title, author, price and qty. Insert a few records.

Setup ODBC

1. Goto "Control Panel" ⇒ Administrator Tools ⇒ Data Source (ODBC),
2. Choose tab "System DSN" (System Data Source Name) (for all users in the system); or "User DSN" (User Data Source Name) (for the current login user only).
3. "Add" ⇒ Select "Microsoft Excel Driver (*.xls, *.xlsx)" (for Excel 2007) ⇒ Click "Finish".
4. In "ODBC Microsoft Access Setup" dialog: In "Data Source Name", enter "`ebookshopODBC`". Click "Select Workbook" and select "`ebookshop.xlsx`" (for Excel 2007) ⇒ OK.

Sample JDBC Program for Excel

```
import java.sql.*;             // Use classes in java.sql package

public class ExcelSelectTest {  // JDK 7 and above
   public static void main(String[] args) {
      try (
         // Step 1: Allocate a database "Connection" object
         Connection conn = DriverManager.getConnection(
               "jdbc:odbc:ebookshopODBC");  // Access/Excel

         // Step 2: Allocate a "Statement" object in the Connection
         Statement stmt = conn.createStatement();
      ) {
         // Excel connection, by default, is read-only.
         // Need to turn it off to issue INSERT, UPDATE, ...
         conn.setReadOnly(false);
         // Step 3: Execute a SQL SELECT query, the query result
         //   is returned in a "ResultSet" object.
         // Table name is the sheet's name in the form of [sheet-name$]
         String strSelect = "select title, price, qty from [books$]";
         System.out.println("The SQL query is: " + strSelect); // Echo For debugging

         ResultSet rset = stmt.executeQuery(strSelect);

         // Step 4: Process the ResultSet by scrolling the cursor forward via next().
         //  For each row, retrieve the contents of the cells with
getXxx(columnName).
         System.out.println("The records selected are:");
         int rowCount = 0;
         while(rset.next()) {   // Move the cursor to the next row
            String title = rset.getString("title");
            double price = rset.getDouble("price");
            int    qty   = rset.getInt("qty");
            System.out.println(title + ", " + price + ", " + qty);
            ++rowCount;
         }
         System.out.println("Total number of records = " + rowCount);

         // Try INSERT
         int returnCode = stmt.executeUpdate(
            "insert into [books$] values (1002, 'Java 101', 'Tan Ah Teck', 2.2, 2)");
         System.out.println(returnCode + " record(s) inserted.");

         // Try UPDATE
         returnCode = stmt.executeUpdate(
            "update [books$] set qty = qty+1 where id = 1002");
         System.out.println(returnCode + " record(s) updated.");

      } catch(SQLException ex) {
         ex.printStackTrace();
      }
      // Step 5: Close the resources - Done automatically by try-with-resources
   }
```

```
}
```

Notes:

1. The `database-URL` is in the form of `jdbc:odbc:{odbc-name}` - the same as Access.
2. The Excel's connection, by default, is read-only. To issue `INSERT|UPDATE|DELETE` commands, you need to disable read-only via `conn.setReadOnly(false)` Otherwise, you will get an error "`java.sql.SQLException: [Microsoft][ODBC Excel Driver] Operation must use an updateable query`".
3. In the SQL command, table name corresponds to sheet name, in the form of `[sheet-name$]`.