1. JDBC – INTRODUCTION

What is JDBC?

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

The JDBC library includes APIs for each of the tasks 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.

Pre-Requisite

Before moving further, you need to have a good understanding of the following two subjects:

- **Core JAVA Programming**
- **SQL or MySQL Database**

JDBC Architecture

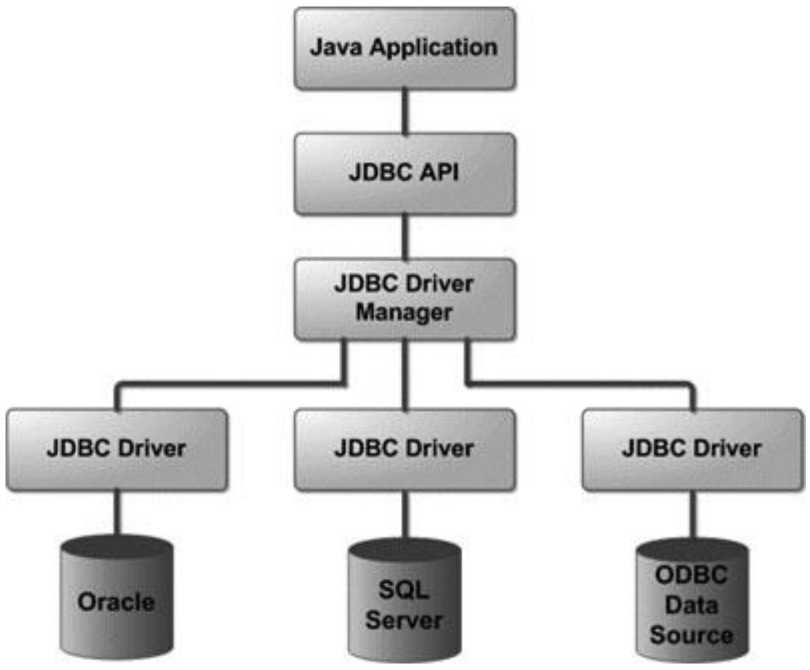
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers:

- **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 subprotocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. 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.

The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. This is the latest JDBC version at the time of writing this tutorial. It offers the main classes for interacting with your data sources.

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

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

2. JDBC – SQL SYNTAX

Structured **Q**uery **L**anguage (SQL) is a standardized language that allows you to perform operations on a database, such as creating entries, reading content, updating content, and deleting entries.

SQL is supported by almost any database you will likely use, and it allows you to write database code independently of the underlying database.

This chapter gives an overview of SQL, which is a prerequisite to understand JDBC concepts. After going through this chapter, you will be able to **C**reate, **R**ead, **U**pdate, and **D**elete (often referred to as **CRUD** operations) data from a database.

For a detailed understanding on SQL, you can read our MySQL Tutorial.

Create Database

The CREATE DATABASE statement is used for creating a new database. The syntax is:

```
SQL> CREATE DATABASE DATABASE_NAME;
```

Example

The following SQL statement creates a Database named EMP:

```
SQL> CREATE DATABASE EMP;
```

Drop Database

The DROP DATABASE statement is used for deleting an existing database. The syntax is:

```
SQL> DROP DATABASE DATABASE_NAME;
```

Note: To create or drop a database you should have administrator privilege on your database server. Be careful, deleting a database would loss all the data stored in the database.

Create Table

The CREATE TABLE statement is used for creating a new table. The syntax is:

```
SQL> CREATE TABLE table_name
(
   column_name column_data_type,
   column_name column_data_type,
   column_name column_data_type
   ...
);
```

Example

The following SQL statement creates a table named Employees with four columns:

```
SQL> CREATE TABLE Employees
(
   id INT NOT NULL,
   age INT NOT NULL,
   first VARCHAR(255),
   last VARCHAR(255),
   PRIMARY KEY ( id )
);
```

Drop Table

The DROP TABLE statement is used for deleting an existing table. The syntax is:

```
SQL> DROP TABLE table_name;
```

Example

The following SQL statement deletes a table named Employees:

```
SQL> DROP TABLE Employees;
```

INSERT Data

The syntax for INSERT, looks similar to the following, where column1, column2, and so on represents the new data to appear in the respective columns:

```
SQL> INSERT INTO table_name VALUES (column1, column2, ...);
```

Example

The following SQL INSERT statement inserts a new row in the Employees database created earlier:

```
SQL> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
```

SELECT Data

The SELECT statement is used to retrieve data from a database. The syntax for SELECT is:

```
SQL> SELECT column_name, column_name, ...
     FROM table_name
     WHERE conditions;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=,and >=, as well as the BETWEEN and LIKE operators.

Example

The following SQL statement selects the age, first and last columns from the Employees table, where id column is 100:

```
SQL> SELECT first, last, age
     FROM Employees
     WHERE id = 100;
```

The following SQL statement selects the age, first and last columns from the Employees table, where *first* column contains *Zara*:

```
SQL> SELECT first, last, age
     FROM Employees
     WHERE first LIKE '%Zara%';
```

UPDATE Data

The UPDATE statement is used to update data. The syntax for UPDATE is:

```
SQL> UPDATE table_name
     SET column_name = value, column_name = value, ...
     WHERE conditions;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=,and >=, as well as the BETWEEN and LIKE operators.

Example

The following SQL UPDATE statement changes the age column of the employee whose id is 100:

```
SQL> UPDATE Employees SET age=20 WHERE id=100;
```

DELETE Data

The DELETE statement is used to delete data from tables. The syntax for DELETE is:

```
SQL> DELETE FROM table_name WHERE conditions;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=,and >=, as well as the BETWEEN and LIKE operators.

Example

The following SQL DELETE statement deletes the record of the employee whose id is 100:

```
SQL> DELETE FROM Employees WHERE id=100;
```

3. JDBC – ENVIRONMENT

To start developing with JDBC, you should setup your JDBC environment by following the steps shown below. We assume that you are working on a Windows platform.

Install Java

Install J2SE Development Kit 5.0 (JDK 5.0) from Java Official Site.

Make sure following environment variables are set as described below:

- **JAVA_HOME:** This environment variable should point to the directory where you installed the JDK, e.g. C:\Program Files\Java\jdk1.5.0.
- **CLASSPATH:** This environment variable should have appropriate paths set, e.g. C:\Program Files\Java\jdk1.5.0_20\jre\lib.
- **PATH:** This environment variable should point to appropriate JRE bin, e.g. C:\Program Files\Java\jre1.5.0_20\bin.

It is possible you have these variable set already, but just to make sure here's how to check.

- Go to the control panel and double-click on System. If you are a Windows XP user, it is possible you have to open Performance and Maintenance, before you will see the System icon.
- Go to the Advanced tab and click on the Environment Variables.
- Now check if all the above mentioned variables are set properly.

You automatically get both JDBC packages **java.sql** and **javax.sql**, when you install J2SE Development Kit 5.0 (JDK 5.0).

Install Database

The most important thing you will need, of course is an actual running database with a table that you can query and modify.

Install a database that is most suitable for you. You can have plenty of choices and most common are:

- **MySQL DB:** MySQL is an open source database. You can download it from **MySQL Official Site**. We recommend downloading the full Windows installation.

  In addition, download and install **MySQL Administrator** as well as **MySQL Query Browser.** These are GUI based tools that will make your development much easier.

Finally, download and unzip **MySQL Connector/J** (the MySQL JDBC driver) in a convenient directory. For the purpose of this tutorial, we will assume that you have installed the driver at C:\Program Files\MySQL\mysql-connector-java-5.1.8.

Accordingly, set CLASSPATH variable to C:\Program Files\MySQL\mysql-connector-java-5.1.8\mysql-connector-java-5.1.8-bin.jar. Your driver version may vary based on your installation.

- **PostgreSQL DB:** PostgreSQL is an open source database. You can download it from **PostgreSQL Official Site**.

  The Postgres installation contains a GUI based administrative tool called pgAdmin III. JDBC drivers are also included as part of the installation.

- **Oracle DB:** Oracle DB is a commercial database sold by Oracle. We assume that you have the necessary distribution media to install it.

  Oracle installation includes a GUI based administrative tool called Enterprise Manager. JDBC drivers are also included as a part of the installation.

Install Database Drivers

The latest JDK includes a JDBC-ODBC Bridge driver that makes most Open Database Connectivity (ODBC) drivers available to programmers using the JDBC API.

Now-a-days, most of the Database vendors are supplying appropriate JDBC drivers along with Database installation. So, you should not worry about this part.

Set Database Credential

For this tutorial we are going to use MySQL database. When you install any of the above database, its administrator ID is set to **root** and gives provision to set a password of your choice.

Using root ID and password you can either create another user ID and password, or you can use root ID and password for your JDBC application.

There are various database operations like database creation and deletion, which would need administrator ID and password.

For rest of the JDBC tutorial, we would use MySQL Database with **username** as ID and **password** as password.

If you do not have sufficient privilege to create new users, then you can ask your Database Administrator (DBA) to create a user ID and password for you.

Create Database

To create the **EMP** database, use the following steps:

Step 1

Open a **Command Prompt** and change to the installation directory as follows:

```
C:\>
C:\>cd Program Files\MySQL\bin
C:\Program Files\MySQL\bin>
```

Note: The path to **mysqld.exe** may vary depending on the install location of MySQL on your system. You can also check documentation on how to start and stop your database server.

Step 2

Start the database server by executing the following command, if it is already not running.

```
C:\Program Files\MySQL\bin>mysqld
C:\Program Files\MySQL\bin>
```

Step 3

Create the **EMP** database by executing the following command:

```
C:\Program Files\MySQL\bin> mysqladmin create EMP -u root -p
Enter password: ********
C:\Program Files\MySQL\bin>
```

Create Table

To create the **Employees** table in EMP database, use the following steps:

Step 1

Open a **Command Prompt** and change to the installation directory as follows:

```
C:\>
C:\>cd Program Files\MySQL\bin
C:\Program Files\MySQL\bin>
```

Step 2

Login to the database as follows:

```
C:\Program Files\MySQL\bin>mysql -u root -p
Enter password: ********
mysql>
```

Step 3

Create the table **Employee** as follows:

```
mysql> use EMP;
mysql> create table Employees
    -> (
    -> id int not null,
    -> age int not null,
    -> first varchar (255),
    -> last varchar (255)
    -> );
Query OK, 0 rows affected (0.08 sec)
mysql>
```

Create Data Records

Finally you create few records in Employee table as follows:

```
mysql> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Employees VALUES (101, 25, 'Mahnaz', 'Fatma');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (102, 30, 'Zaid', 'Khan');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (103, 28, 'Sumit', 'Mittal');
Query OK, 1 row affected (0.00 sec)

mysql>
```

For a complete understanding on MySQL database, study the MySQL Tutorial.

Now you are ready to start experimenting with JDBC. Next chapter gives you a sample example on JDBC Programming.

4. JDBC – SAMPLE CODE

This chapter provides an example of how to create a simple JDBC application. This will show you how to open a database connection, execute a SQL query, and display the results.

All the steps mentioned in this template example, would be explained in subsequent chapters of this tutorial.

Creating JDBC Application

There are following six steps involved in building a JDBC application:

- **Import the packages**: Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- **Register the JDBC driver**: Requires that you initialize a driver so you can open a communication channel with the database.
- **Open a connection**: Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
- **Execute a query**: Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Extract data from result set**: Requires that you use the appropriate*ResultSet.getXXX()* method to retrieve the data from the result set.
- **Clean up the environment**: Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

Sample Code

This sample example can serve as a **template** when you need to create your own JDBC application in the future.

This sample code has been written based on the environment and database setup done in the previous chapter.

Copy and paste the following example in FirstExample.java, compile and run as follows:

```
//STEP 1. Import required packages
import java.sql.*;


public class FirstExample {
```

```
// JDBC driver name and database URL
static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
static final String DB_URL = "jdbc:mysql://localhost/EMP";

//  Database credentials
static final String USER = "username";
static final String PASS = "password";

public static void main(String[] args) {
Connection conn = null;
Statement stmt = null;
try{
   //STEP 2: Register JDBC driver
   Class.forName("com.mysql.jdbc.Driver");

   //STEP 3: Open a connection
   System.out.println("Connecting to database...");
   conn = DriverManager.getConnection(DB_URL,USER,PASS);

   //STEP 4: Execute a query
   System.out.println("Creating statement...");
   stmt = conn.createStatement();
   String sql;
   sql = "SELECT id, first, last, age FROM Employees";
   ResultSet rs = stmt.executeQuery(sql);
   //STEP 5: Extract data from result set
   while(rs.next()){
      //Retrieve by column name
      int id  = rs.getInt("id");
      int age = rs.getInt("age");
      String first = rs.getString("first");
      String last = rs.getString("last");
      //Display values
      System.out.print("ID: " + id);
```

```
         System.out.print(", Age: " + age);
         System.out.print(", First: " + first);
         System.out.println(", Last: " + last);
      }
      //STEP 6: Clean-up environment
      rs.close();
      stmt.close();
      conn.close();
   }catch(SQLException se){
      //Handle errors for JDBC
      se.printStackTrace();
   }catch(Exception e){
      //Handle errors for Class.forName
      e.printStackTrace();
   }finally{
      //finally block used to close resources
      try{
         if(stmt!=null)
            stmt.close();
      }catch(SQLException se2){
      }// nothing we can do
      try{
         if(conn!=null)
            conn.close();
      }catch(SQLException se){
         se.printStackTrace();
      }//end finally try
   }//end try
   System.out.println("Goodbye!");
}//end main
}//end FirstExample
```

Now let us compile the above example as follows:

```
C:\>javac FirstExample.java
```

```
C:\>
```

When you run **FirstExample**, it produces the following result:

```
C:\>java FirstExample
Connecting to database...
Creating statement...
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
C:\>
```

5. JDBC – DRIVER TYPES

What is JDBC Driver?

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.

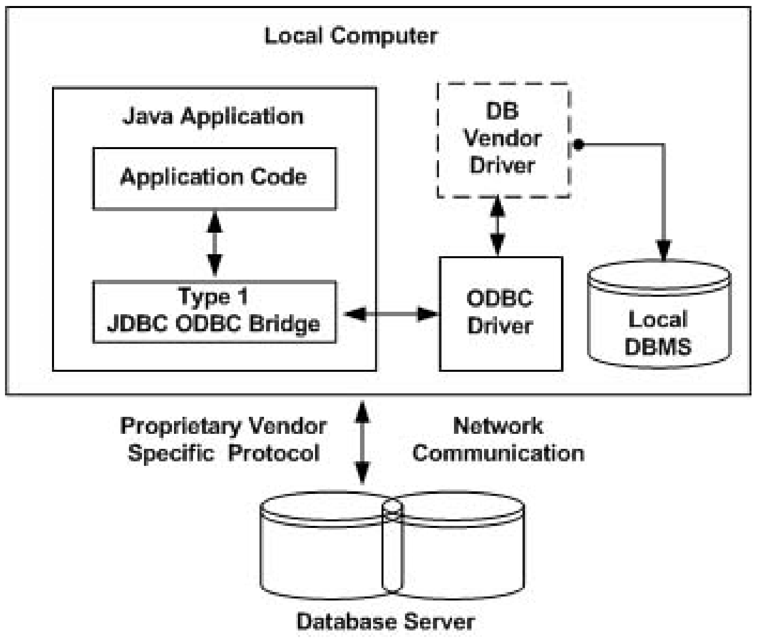
JDBC Drivers Types

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:

Type 1: JDBC-ODBC Bridge Driver

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.

When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.

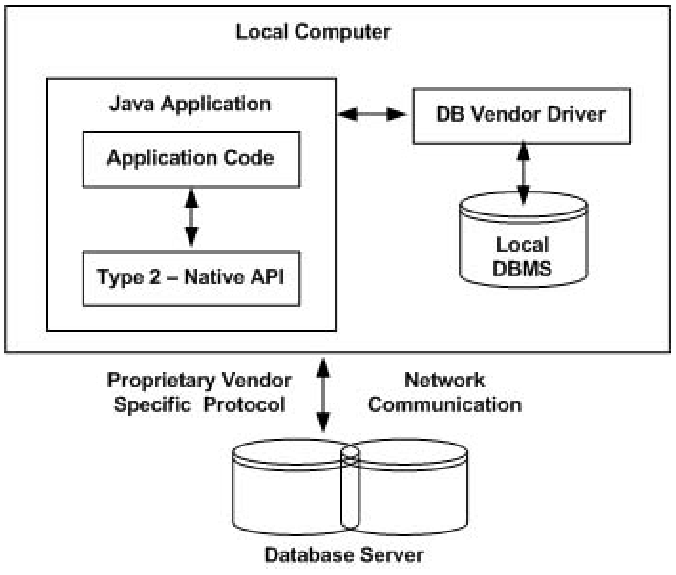


The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

Type 2: JDBC-Native API

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.

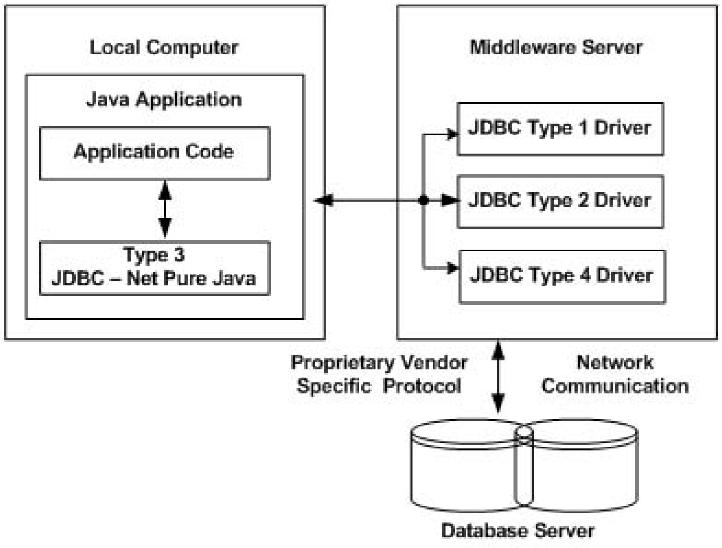


The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

Type 3: JDBC-Net Pure Java

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.



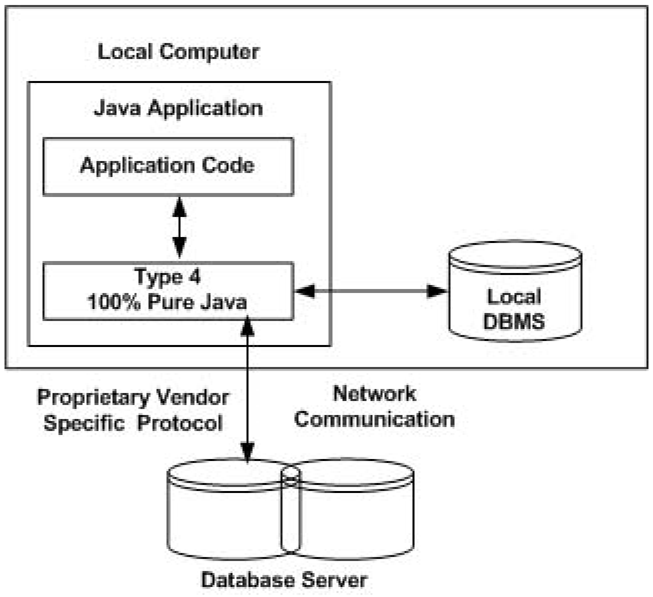
You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

Type 4: 100% Pure Java

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.



MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

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.

6. JDBC – CONNECTIONS

After you've installed the appropriate driver, it is time to establish a database connection using JDBC.

The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps:

- **Import JDBC Packages:** Add **import** statements to your Java program to import required classes in your Java code.
- **Register JDBC Driver:** This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
- **Database URL Formulation:** This is to create a properly formatted address that points to the database to which you wish to connect.
- **Create Connection Object:** Finally, code a call to the *DriverManager* object's *getConnection()* method to establish actual database connection.

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

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:@hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname: port Number/databaseName

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

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);
```