Introduction

A database is a collection of information that is organized so that it can be easily accessed, managed and updated.

Data is organized into rows, columns and tables, and it is indexed to make it easier to find relevant information. Data gets updated, expanded and deleted as new information is added. Databases process workloads to create and update themselves, querying the data they contain and running applications against it. This process is known as DBMS, as it allows mechanism for storing, organizing, retrieving and modifying data for many users.

Core JDBC Components

JDBC Drivers

A JDBC driver is a collection of Java classes that enables you to connect to a certain database. For instance, MySQL will have its own JDBC driver. A JDBC driver implements a lot of the JDBC interfaces. When your code uses a given JDBC driver, it actually just uses the standard JDBC interfaces. The concrete JDBC driver used is hidden behind the JDBC interfaces. Thus you can plugin a new JDBC driver without your code noticing it.

Of course, the JDBC drivers may vary a little in the features they support.

Connections

Once a JDBC driver is loaded and initialized, you need to connect to the database. You do so by obtaining a Connection to the database via the JDBC API and the loaded driver. All communication with the database happens via a connection. An application can have more than one connection open to a database at a time. This is actually very common.

Statements

A Statement is what you use to execute queries and updates against the database. There are a few different types of statements you can use. Each statement corresponds to a single query or update.

ResultSets

When you perform a query against the database you get back a ResultSet. You can then traverse this ResultSet to read the result of the query.

Common JDBC Use Cases

Query the database

One of the most common use cases is to read data from a database. Reading data from a database is called querying the database.

Query the database meta data

Another common use case is to query the database meta data. The database meta data contains information about the database itself. For instance, information about the tables defined, the columns in each table, the data types etc.

Update the database

Another very common JDBC use case is to update the database. Updating the database means writing data to it. In other words, adding new records or modifying (updating) existing records.

Perform transactions

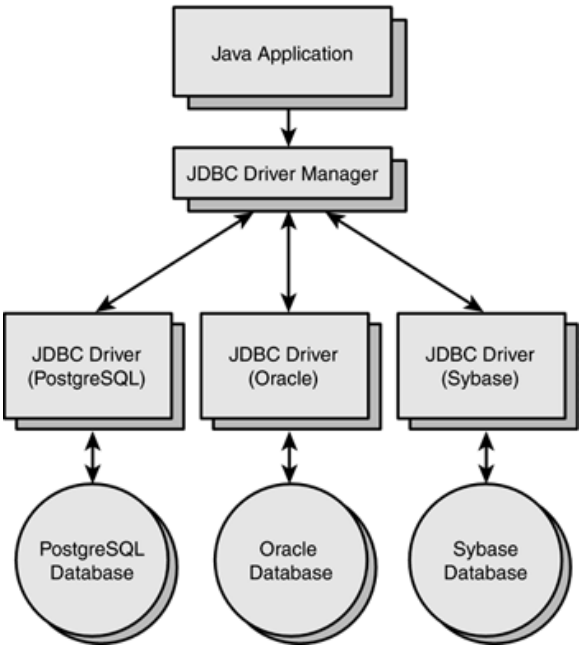
Transactions is anoter common use case. A transaction groups multiple updates and possibly queries into a single action. Either all of the actions are executed, or none of them are.

JDBC Architecture

JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database. JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

JDBC is similar in structure to ODBC. A JDBC application is composed of multiple layers, as shown in Figure 1.

Figure 1. JDBC architecture.



The topmost layer in this model is the Java application. Java applications are portable?you can run a Java application without modification on any system that has a Java runtime environment installed. A Java

application that uses JDBC can talk to many databases with few, if any, modifications. Like ODBC, JDBC provides a consistent way to connect to a database, execute commands, and retrieve the results. Also like ODBC, JDBC does not enforce a common command language?you can use Oracle-specific syntax when connected to an Oracle server and PostgreSQL-specific syntax when connected to a PostgreSQL server.

The JDBC DriverManager

The JDBC DriverManager class is responsible for locating a JDBC driver needed by the application. When a client application requests a database connection, the request is expressed in the form of a URL (Uniform Resource Locator). A typical URL might look like jdbc:postgresql:movies.

The JDBC Driver

As each driver is loaded into a Java Virtual Machine (VM), it registers itself with the JDBC DriverManager. When an application requests a connection, the DriverManager asks each Driver whether it can connect to the database specified in the given URL. As soon as it finds an appropriate Driver, the search stops and the Driver attempts to make a connection to the database. If the connection attempt fails, the Driver will throw a SQLException to the application. If the connection completes successfully, the Driver creates a Connection object and returns it to the application.

The JDBC-Compliant Database

The bottom layer of the JDBC model is the database. The PostgreSQL Driver class (and other JDBC classes) translates application commands into PostgreSQL network requests and translates the results back into JDBC object form.

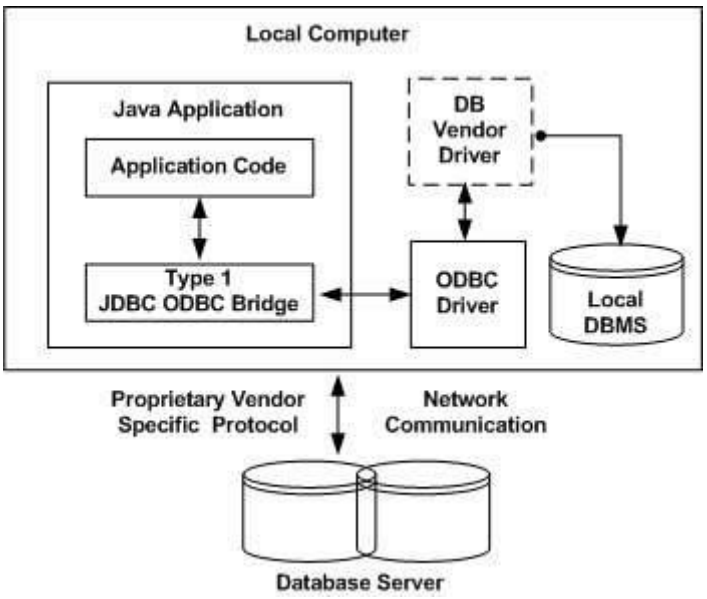
JDBC Drivers Types

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.

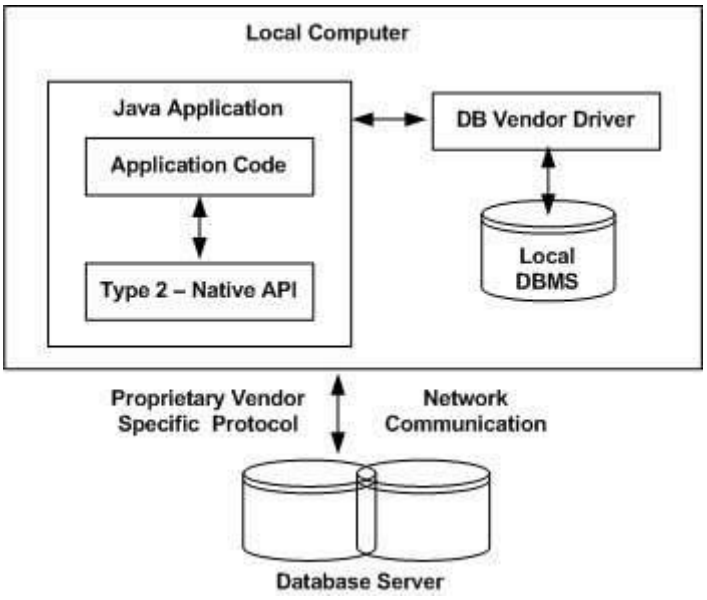


The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

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.

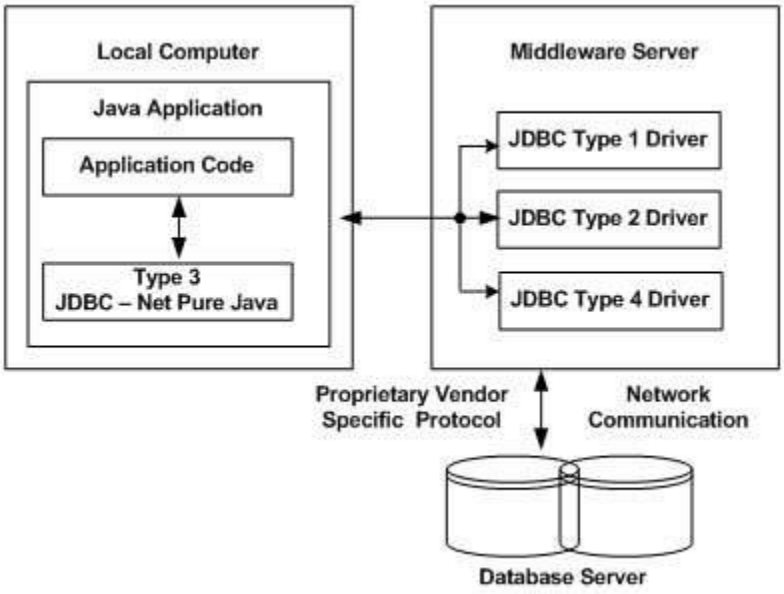


The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

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.



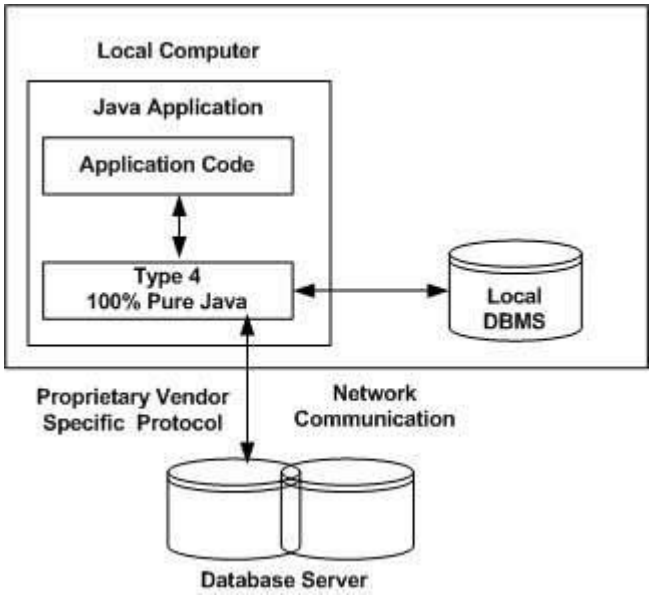
You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

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.

(Alternative → you can write them as these too

• A type 1 driver translates JDBC to ODBC and relies on an ODBC driver to communicate with the database. Early versions of Java included one such driver, the JDBC/ODBC bridge. However, the bridge requires deployment and proper configuration of an ODBC driver. When JDBC was first released, the bridge was handy for testing, but it was never intended for production use. At this point, many better drivers are available, and we advise against using the JDBC/ODBC bridge.•

A type 2 driver is written partly in Java and partly in native code; it communicates with the client API of a database. When using such a driver, you must install some platform-specific code onto the client in addition to a Java library.

• A type 3 driver is a pure Java client library that uses a database-independent protocol to communicate database requests to a server component, which then translates the requests into a database-specific protocol. This simplifies deployment because the platform-specific code is located only on the server.

• A type 4 driver is a pure Java library that translates JDBC requests directly to a database-specific protocol.)

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.

Typical Uses Of JDBC

Client/server architecture means an application that connects directly to a database. Business logic and the user interface are implemented together on the client. Your application may also have business logic as stored procedures in the database.

In Java applications, the database connection is made using JDBC. The traditional client/server model has a rich GUI on the client and a database on the server (see Figure 2). In this model, a JDBC driver is deployed on the client.

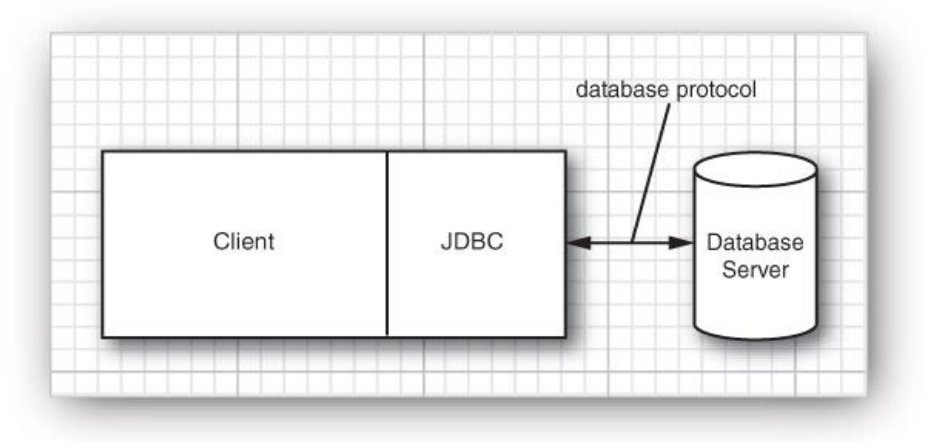


Figure 2. A traditional client/server application

However, the world is moving away from client/server and toward a three-tier model or even more advanced *n*-tier models. In the three-tier model, the client does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a couple of advantages. It separates *visual presentation* (on the client) from the *business logic* (in the middle tier) and the raw data (in the database). Therefore, it becomes possible to access the same data and the same business rules from multiple clients, such as a Java application, an applet, or a web form.

Communication between the client and the middle tier can occur through HTTP (when you use a web browser as the client) or another mechanism such as remote method. JDBC manages the communication between the middle tier and the back-end database. Figure 2 shows the basic architecture. There are, of course, many variations of this model. In particular, the Java

Enterprise Edition defines a structure for *application servers* that manage code modules called *Enterprise JavaBeans*, and provides valuable services such as load balancing, request caching, security, and object-relational mapping. In that architecture, JDBC still plays an important role for issuing complex database queries.

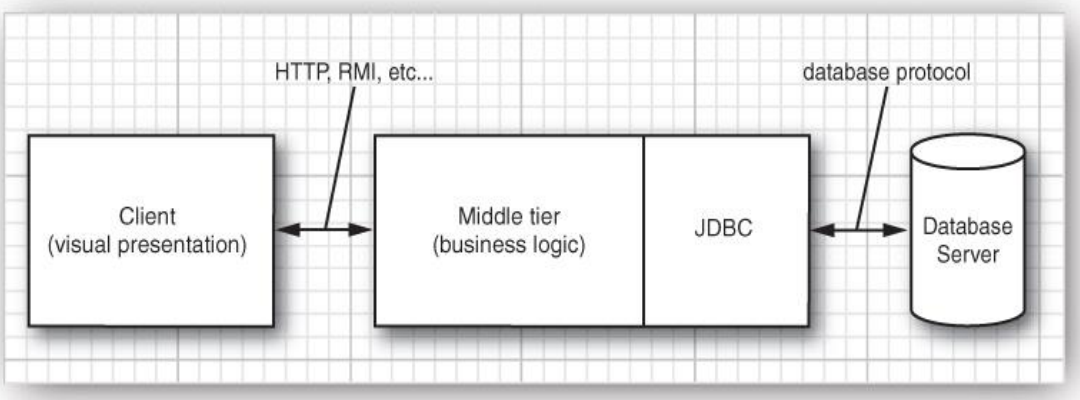


Figure 2. A three-tier application

Manipulating databases with JDBC

Java programs communicate with the database and manipulate its data with the help of the JDBC API. The JDBC driver enables the Java application to connect to a database. JDBC is almost always used with relational databases, also it can be used with any other table based data source. We do not have to worry about the availability of a driver, as major RDBMS (Relational Database Management System) providers provide them free. Apart from that there are many third-party JDBC drivers available.

Basic Requirements

Since we shall be going hands on down the line, the basic software requirements for JDBC programming are as follows.

1.Java SDK

2.RDBMS Package (For example, MySQL, Oracle, PostgreSQL, etc.)

3.IDE (For example, Eclipse, NetBeans, JDeveloper, etc.)

4.JDBC driver (JDBC drivers are database specific, especially, if we use a driver other than Type1:JDBC-ODBC Bridge. For example, MySQL Connector/J is the official JDBC driver for MySQL, ojdbc for Oracle and so on...PostgreSQL JDBC Driver)

Installation is pretty straightforward; if in doubt, refer to the appropriate installation instruction of the relevant packages during installation.

JDBC Programming Steps

Every Java code in JDBC Programming goes through the following six steps in one way or the other. These steps give an idea about what order to follow during coding and a basic insight into their significance.

1. **Importing *java.sql* Package**

Almost all the classes and interfaces used in JDBC programming are compiled in the *java.sql* package. As a result it is our primary requirement to import the package as follows.

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

2. **Load and Register JDBC Driver**

The most common and easiest way to load the driver is by using the *Class.forName()* method.

```
Class.forName("com.mysql.jdbc.Driver");
```

This method takes the complete package name of the driver as its argument. Once the driver is loaded, it will call the *DriverManager.registerDriver()* method to register itself. Registering a driver implies that the currently registered driver is added to a list of available *Driver* objects maintained by the *DriverManager*. The driver manager serves the request from the application using one of the lists of available *Driver* objects.

3. **Establishing Connection**

The standard method to establish a connection to a database is through the method call *DriverManager.getConnection()*. The arguments accepted by this method are: a string representation of the database URL, the user name to log in to the database and the password.

```
DriverManager.getConnection("jdbc:mysql://localhost/hr","user1","pass");
```

4. **Creating a Statement**

We need to create a *Statement* object to execute a static SQL query. Static SQL statements can be updates, inserts, queries and even DDL SQL statements. *Statement* objects are created with the help of the *Connection* object's *createStatement()* method as follows:

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

5. **Execute SQL Statement and Retrieve Result**

We can use the *executeQuery()* method of the *Statement* object to fire the query to the database. This method takes an SQL query string as an argument and returns the result as a *ResultSet* object. The *ResultSet* object contains both the data returned by the query and methods for retrieving the data.

```
ResultSet resultSet=statement.executeQuery("SELECT
* FROM employees");
```

The get methods of the *ResultSet* object can be used to retrieve each of the fields in the record fetched from the database into java variables.

```
while(resultSet.next()){
      System.out.println(resultSet.getString("emp_id"));
      System.out.println(resultSet.getString("first_name"));
      System.out.println(resultSet.getString("last_name"));
      ...
      }
```

6. **Close Connection**

It is highly recommended that an application should close the *Connection* object and *Statement* objects explicitly, because, earlier opened connections can cause trouble for the database and open connections are also prone to security threats. Simply add following statements:

```
statement.close();
connection.close();
```

Putting it Together

Accessing Data with JDBC

Without a doubt the data access is one of the most used features in development. Practically the whole system needs a database, with rare exceptions.

Java, different than languages like PHP does not support access to the database directly, so that it uses an API (group of classes and interfaces) to do the job. The JDBC (Java Database Connectivity) make the sending of SQL statements to any relational database, provided there is a driver that matches the same gift.

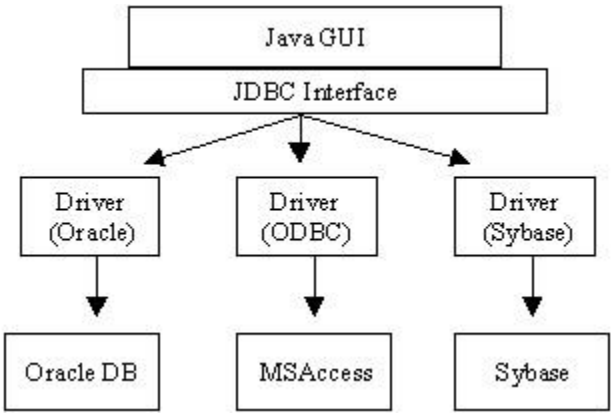


Figure 1: Schematic of operation of the JDBC

There are four types of JDBC drivers: 1, 2, 3 and 4, they are:

Type 1: JDBC-ODBC Bridge

It is the simplest type but restricted to the Windows platform. Uses ODBC to connect to the database, converting methods in JDBC calls to ODBC functions. This bridge is typically used when there is a pure-Java driver (type 4) for a given database, because its use is discouraged due to the dependence of the platform.

Type 2: Native-API Driver

The Native-API driver translates JDBC calls to the API calls customer database used. As the JDBC-ODBC Bridge, may need extra software installed on the client machine.

Type 3: Driver Network Protocol

Translates the JDBC call to a network protocol independent of the database used, which is translated into the protocol database for a server. By using an independent protocol, clients can connect Java applications to several different databases. It's more flexible model.

Type 4: Native Driver

Converts JDBC calls directly into the protocol database. Implemented in Java, normally is independent platform and written by the developers themselves. It is the most recommended to be used.

In this article we use the type 4 being the most recommended. Because he takes calls directly in the protocol of the database in question, thus giving better performance, besides, of course, being the simplest to use.

Many may find a certain similarity between JDBC and ODBC, and are absolutely correct, we can say "roughly" the two follow the same idea. Both work as a communication application x

Bank, but ODBC is a Windows application restricted to it, while the JDBC because it is written in java, is multiplatform.

Another advantage of JDBC is that it act as a data abstraction layer. Independent of SGBD used, the API is the same, greatly facilitating the life of programmers if there is a need for a database migration.

Now we will see how to connect through JDBC in a Oracle database.

Listing 1: Connecting to an Oracle database

```
Connection connection = null;
    try {
        // Load the JDBC driver
        String driverName = "oracle.jdbc.driver.OracleDriver";
        Class.forName(driverName);

        // Create a connection to the database
        String serverName = "127.0.0.1";
        String portNumber = "1521";
        String sid = "mydatabase";
        String url = "jdbc:oracle:thin:@" + serverName + ":" + portNumber +
":" + sid;
        String username = "username";
        String password = "password";
        connection = DriverManager.getConnection(url, username, password);
    } catch (ClassNotFoundException e) {
        // Could not find the database driver
    } catch (SQLException e) {
        // Could not connect to the database
    }
```

The code is well standardized, we put the server ip, port number, database name, username and password, and put the driver connection.

Now we will see how to access a database in MySQL.

Listing 2: Accessing a MySQL database

```
Connection connection = null;
    try {
        // Load the JDBC driver
        String driverName = "org.gjt.mm.mysql.Driver"; // MySQL MM JDBC
driver
        Class.forName(driverName);

        // Create a connection to the database
        String serverName = "localhost";
        String mydatabase = "mydatabase";
        String url = "jdbc:mysql://" + serverName +  "/" + mydatabase; // a
JDBC url
        String username = "username";
```

```
        String password = "password";
        connection = DriverManager.getConnection(url, username, password);
    } catch (ClassNotFoundException e) {
        // Could not find the database driver
    } catch (SQLException e) {
        // Could not connect to the database
    }
```

Note that the difference of the code in Listing 1 to 2 is just the name of the driver that changes jdbc:oracle for jdbc:mysql and its application too.

How to load the JDBC driver

Now we will see how to load a JDBC driver, in this example we will do load the MySQL driver.

Listing 3: Load the JDBC driver

```
  try {
        // Load the JDBC driver
        String driverName = "org.gjt.mm.mysql.Driver";
        Class.forName(driverName);
    } catch (ClassNotFoundException e) {
        // Could not find the driver
    }
```

Let us understand better now what each command means.

Class.forName - This is without doubt the main command. It is through him that we are calling the JDBC driver.

Connection - Here we are creating an object of type "Connection". This is where the information is stored in your database connection. To be more direct, we use the method "getConnection" of the object "DriverManager" contained in "java.sql" instead of the default constructor, this causes the connection is established immediately. Note the string passed as parameter, the information contained in it are our connection, they are respectively: JDBC driver, host, database path, username and password for last.

Statement - a statement is simple, but vital for any project. The object "Statement" is responsible for receiving commands and SQL to relay information, as well as the return.

Now that we know how to create the connection and loading the driver, what do you think of seeing some basic operations?

Deleting all records from the table

We will see now how to delete all records of a table. Be very careful with this code because it will clear all records of your table.

Listing 4: Deleting all records with JDBC

```
try {
        Statement stmt = connection.createStatement();

        // Use TRUNCATE
        String sql = "TRUNCATE my_table";

        // Use DELETE
        sql = "DELETE FROM my_table";

        // Execute deletion
        stmt.executeUpdate(sql);
    } catch (SQLException e) {
    }
```

As we know delete all records is very dangerous and little used, the next listing we will see how to delete only one record.

Listing 5: Excluding only one record with JDBC

```
try {
        // Create a statement
        Statement stmt = connection.createStatement();

        // Prepare a statement to insert a record
        String sql = "DELETE FROM my_table WHERE col_string='a string'";

        // Execute the delete statement
        int deleteCount = stmt.executeUpdate(sql);
        // deleteCount contains the number of deleted rows

        // Use a prepared statement to delete

        // Prepare a statement to delete a record
        sql = "DELETE FROM my_table WHERE col_string=?";
        PreparedStatement pstmt = connection.prepareStatement(sql);
        // Set the value
        pstmt.setString(1, "a string");
        deleteCount = pstmt.executeUpdate();
        System.err.println(e.getMessage());
} catch (SQLException e) {
    }
```

Thus we can only delete one record, one that we passed as parameter.

Inserting records with JDBC

If we want to insert a record into a database table is very simple, just insert the following code.

Listing 6: Inserting data into the database

```
try {
```

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

        // Prepare a statement to insert a record
        String sql = "INSERT INTO my_table (col_string) VALUES('a string')";

        // Execute the insert statement
        stmt.executeUpdate(sql);
    } catch (SQLException e) {
    }
```

Updating records

If you want to edit or update a table record is also very simple, just we use the code in Listing 7.

Listing 7: Changing records

```
try {
        Statement stmt = connection.createStatement();

        // Prepare a statement to update a record
        String sql = "UPDATE my_table SET col_string='a new string' WHERE
col_string = 'a string'";

        // Execute the insert statement
        int updateCount = stmt.executeUpdate(sql);
        // updateCount contains the number of updated rows
    } catch (SQLException e) {
    }
```

TYPES OF STATEMENTS

There are 3 types of Statements, as given below:

Statement:

It can be used for general-purpose access to the database. It is useful when you are using static SQL statements at runtime.

PreparedStatement:

It can be used when you plan to use the same SQL statement many times. The PreparedStatement interface accepts input parameters at runtime.

CallableStatement:

CallableStatement can be used when you want to access database stored procedures.

ODBC(Open Database Connectivity)

Open Database Connectivity (ODBC) is an open standard application programming interface (API) that allows application programmers to access any database.

	ODBC	**JDBC**
Definition	ODBC - Open DataBase Connectivity.	JDBC - Java Database Connectivity.
Languages	Can be used for C/C++/java.	Only For Java.
Platform	Windows and uses dlls	Most of the platforms being cross-platform compatibility and uses jars.
Driver Development	ODBC drivers are implemented in native languages like C/C++.	JDBC drivers are implemented in Java.
Performance	ODBC drivers are faster.	JDBC drivers are slower than ODBC drivers.
Type	ODBC is Procedural	JDBC is object oriented.
Limitations	ODBC can't be directly used with Java because it uses a C interface.	It is therefore designed especially for java.
Installation	ODBC requires manual installation of the ODBC driver manager and driver on all client machines.	JDBC code is automatically installable, secure, and portable on all platforms.
Dependency	Language Independent	Language Dependent.

Table: difference of JDBC and ODBC

ODBC is a software API method for using database management systems (DBMS). ODBC was created so as to be independent of programming languages or operational systems and offers access to different database systems. The standard ODBC consists of an ODBC core and the respective specific ODBC database drivers. The core, also known as Driver manager, is independent of the database and acts as an interpreter between the application and the database drivers. The database drivers, on the other hand, contain DBMS-specific details and offer a mechanism for connecting with different ODBS-enabled database systems.