

JDBC

JAVA DATABASE CONNECTIVITY

HAZIRLAYAN AYTEKİN KAPLAN



## JDBC Tutorial

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.

### Why to Learn JDBC?

JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases. The JDBC library includes APIs for 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.

### Applications of JDBC

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.

### 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.

### Audience

This tutorial is designed for Java programmers who would like to understand the JDBC framework in detail along with its architecture and actual usage.

### Prerequisites

Before proceeding with this tutorial, you should have a good understanding of Java programming language. As you are going to deal with RDBMS, you should have prior exposure to SQL and Database concepts.

## JDBC – Introduction

### What is JDBC?

JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for 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.

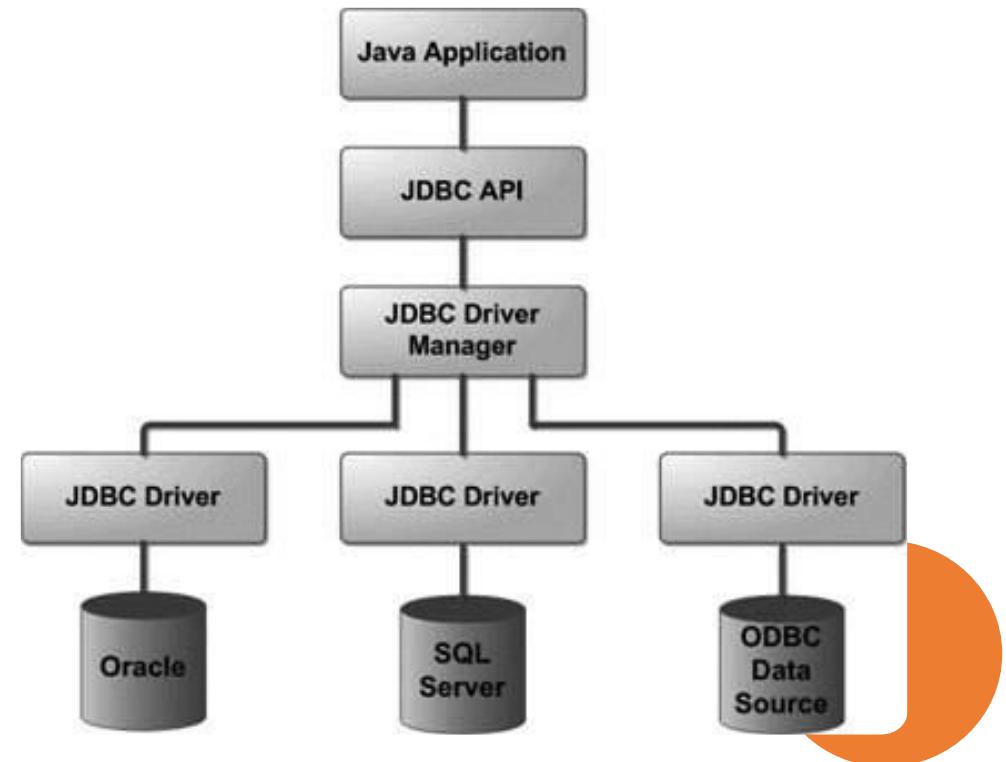
JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

### 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 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.

## 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.

# JDBC - SQL Syntax

Structured Query Language (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 Create, Create, Read, Update, and Delete (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;
```

# JDBC - Environment Setup

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

Java SE is available for download for free. To download [click here](#), please download a version compatible with your operating system.

Follow the instructions to download Java, and run the .exe to install Java on your machine. Once you have installed Java on your machine, you would need to set environment variables to point to correct installation directories.

### Setting Up the Path for Windows 2000/XP

Assuming you have installed Java in c:\Program Files\java\jdk directory –

- Right-click on 'My Computer' and select 'Properties'.
- Click on the 'Environment variables' button under the 'Advanced' tab.
- Now, edit the 'Path' variable and add the path to the Java executable directory at the end of it. For example, if the path is currently set to C:\Windows\System32, then edit it the following way

```
C:\Windows\System32;c:\Program Files\java\jdk\bin
```

### Setting Up the Path for Windows 95/98/ME

Assuming you have installed Java in c:\Program Files\java\jdk directory –

- Edit the 'C:\autoexec.bat' file and add the following line at the end –

```
SET PATH = %PATH%;C:\Program Files\java\jdk\bin
```

### Setting Up the Path for Linux, UNIX, Solaris, FreeBSD

Environment variable PATH should be set to point to where the Java binaries have been installed. Refer to your shell documentation if you have trouble doing this.

For example, if you use bash as your shell, then you would add the following line at the end of your .bashrc –

```
export PATH = /path/to/java:$PATH
```

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

## 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.

## Set Database Credential

When we install MySQL database, its administrator ID is set to **root** and it 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 **guest** as ID and **guest123** 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 **TUTORIALSPOINT** 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 **TUTORIALSPOINT** database by executing the following command –

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

## Create Table

To create the **Employees** table in TUTORIALSPOINT 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 guest -p  
Enter password: *****  
mysql>
```

### Step 3

Create the table **Employees** as follows –

```
mysql> use TUTORIALSPOINT;  
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.

```
1 import java.sql.*;
2
3 new *
4 public class FirstConnectionAttempt09 {
5     new *
6     public static void main(String[] args) {
7         String url = "jdbc:mysql://localhost:3306/newbase04";
8         String userName = "root";
9         String password = "1234";
10        String query = "SELECT * FROM products";
11        try (Connection connection = DriverManager.getConnection(url, userName, password);
12             Statement statement = connection.createStatement();
13             ResultSet resultSet = statement.executeQuery(query);) {
14            while (resultSet.next()) {
15                System.out.println(resultSet.getInt(columnLabel: "id"));
16                System.out.println(resultSet.getString(columnLabel: "name"));
17                System.out.println(resultSet.getDouble(columnLabel: "price"));
18                System.out.println(resultSet.getString(columnLabel: "image_url"));
19                System.out.println();
20            }
21        } catch (SQLException e) {
22            e.printStackTrace();
23        }
24    }
}
```

```
ID : 100 NAME: Samsung S5 PRICE: 2000.0 IMAGE: 1.jpg
ID : 101 NAME: Samsung S6 PRICE: 3000.0 IMAGE: 2.jpg
ID : 104 NAME: IPhone 8 PRICE: 4000.0 IMAGE: 3.jpg
ID : 105 NAME: IPhone 12 PRICE: 4500.0 IMAGE: 4.jpg
ID : 106 NAME: IPhone 12 Pro Max PRICE: 5000.0 IMAGE: 5.jpg
ID : 107 NAME: IPhone 13 PRICE: 5500.0 IMAGE: 6.jpg
ID : 108 NAME: IPhone 13 Pro Max PRICE: 6000.0 IMAGE: 7.jpg
ID : 109 NAME: IPhone 14 PRICE: 6250.0 IMAGE: 8.jpg
ID : 110 NAME: IPhone 14 Pro PRICE: 6500.0 IMAGE: 9.jpg
ID : 111 NAME: IPhone Pro Max PRICE: 7000.0 IMAGE: 10.jpg
```

# 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 implementations are done in third-party drivers. Third party vendors implements the *java.sql.Driver* interface in their database driver.

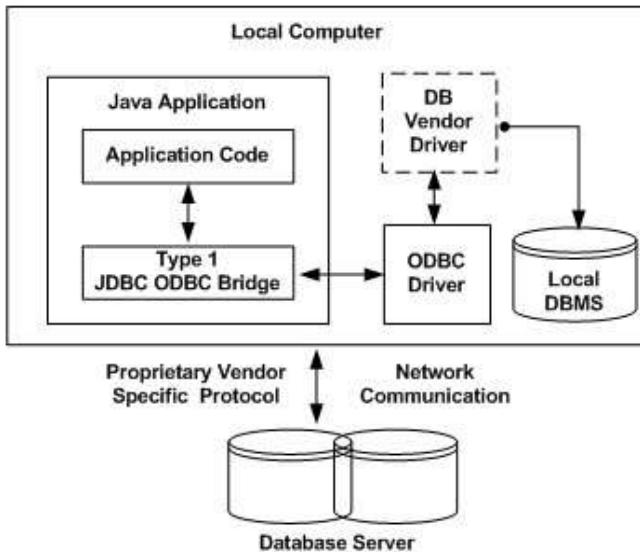
## JDBC Drivers Types

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.

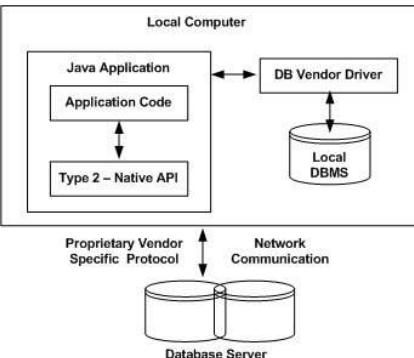


The **JDBC-ODBC Bridge** that comes with JDK 1.2 is a good example of this kind of driver.

## 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.

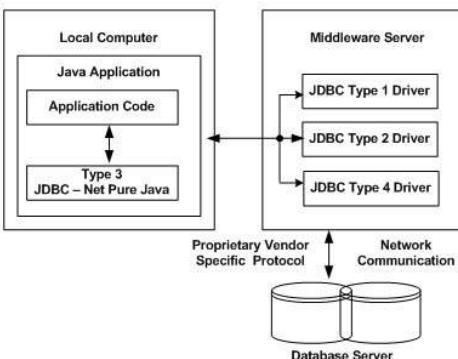


The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

## 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.



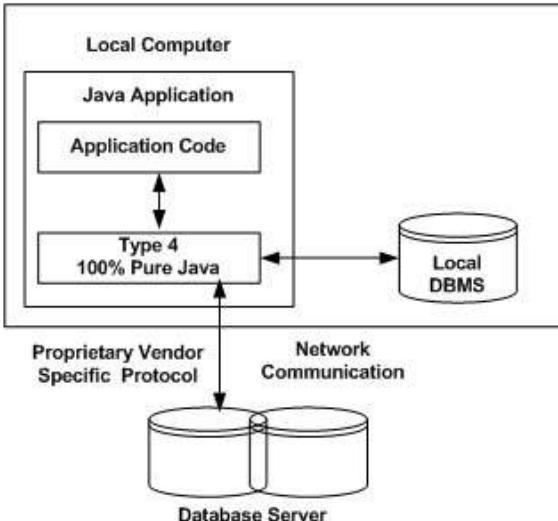
You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

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.

# JDBC - Database 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	<code>jdbc:mysql://hostname/ databaseName</code>
ORACLE	oracle.jdbc.driver.OracleDriver	<code>jdbc:oracle:thin:@hostname:port Number:databaseName</code>
DB2	COM.ibm.db2.jdbc.net.DB2Driver	<code>jdbc:db2:hostname:port Number/databaseName</code>
Sybase	com.sybase.jdbc.SybDriver	<code>jdbc:sybase:Tds:hostname: port Number/databaseName</code>

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 `getConneciton()` 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.

For a better understanding, we suggest you to study our [JDBC - Sample Code tutorial](#).

# JDBC - Statements, PreparedStatement and CallableStatement

Once a connection is obtained we can interact with the database. The JDBC *Statement*, *CallableStatement*, and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

## The Statement Objects

### Creating Statement Object

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's `createStatement()` method, as in the following example –

```
Statement stmt = null;
try {
    stmt = conn.createStatement( );
    ...
}
catch (SQLException e) {
    ...
}
finally {
    ...
}
```

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

- **boolean execute (String SQL):** Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate (String SQL)** – Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL)** – Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

## Closing Statement Object

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

```
Statement stmt = null;
try {
    stmt = conn.createStatement();
    . .
}
catch (SQLException e) {
    . .
}
finally {
    stmt.close();
}
```

For a better understanding, we suggest you to study the [Statement - Example tutorial](#).

## The PreparedStatement Objects

The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

## Creating PreparedStatement Object

```
PreparedStatement pstmt = null;
try {
    String SQL = "Update Employees SET age = ? WHERE id = ?";
    pstmt = conn.prepareStatement(SQL);
    . .
}
catch (SQLException e) {
    . .
}
finally {
    . .
}
```

All parameters in JDBC are represented by the ? symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

All of the **Statement object's** methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

id si 5 olan  
elemanın  
yaşı  
güncellenm  
eden önce

```
1 •  create table employees
2 •  (id int primary key auto_increment,
3 •      first_name varchar(50),
4 •      last_name varchar(50),
5 •      age int
6 • );
7
8 •  insert into employees(first_name, last_name, age) va
9 ("Saim", "Ayhan", 30),
10 ("Altuğ", "Paylan", 32),
11 ("Nuri", "Zor", 34),
12 ("Alper", "Tunga", 40),
13 ("Boğaç", "Han", 45),
14 ("Orçun", "Zaloğlu", 48),
15 ("Efkan", "Özür", 59),
16 ("Süleyman", "İbret", 50),
17 ("Erman", "Hantallı", 29),
18 ("Ersan", "Şamar", 39),
19 ("Buket", "Kal", 47);
20
21 •  select * from employees;
22
23
```

Result Grid | Filter Rows: Edit: Export/Import: Wrap Cell Content:

	id	first_name	last_name	age
1	Saim	Ayhan	30	
2	Altuğ	Paylan	32	
3	Nuri	Zor	34	
4	Alper	Tunga	40	
5	Boğaç	Han	45	
6	Orçun	Zaloğlu	48	
7	Efkan	Özür	59	
8	Süleyman	İbret	50	
9	Erman	Hantallı	29	
10	Ersan	Şamar	39	
11	Buket	Kal	47	
	NULL	NULL	NULL	NULL

# İd si 5 olan elemanın yaşı güncellendi

```
new *  
public class FirstConnectionAttempt10 {  
    new *  
    public static void main(String[] args) {  
        String url = "jdbc:mysql://localhost:3306/newbase21";  
        String userName = "root";  
        String password = "1234";  
        String query = "Select id, first_name, last_name, age from employees";  
        String updateQuery = "UPDATE employees set age=21 where id=5";  
  
        try (Connection connection = DriverManager.getConnection(url, userName, password);  
             Statement statement = connection.createStatement();) {  
  
            System.out.println("Return value is : " + statement.toString());  
            System.out.println("Row is impacted: " + statement.executeUpdate(updateQuery));  
  
            ResultSet resultSet = statement.executeQuery(query);  
  
            while (resultSet.next()) {  
                System.out.print("ID: " + resultSet.getInt(columnLabel: "id"));  
                System.out.print(", Age: " + resultSet.getInt(columnLabel: "age"));  
                System.out.print(", First Name: " + resultSet.getString(columnLabel: "first_name"));  
                System.out.println(", Lastname: " + resultSet.getString(columnLabel: "last_name"));  
            }  
            resultSet.close();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
Return value is : com.mysql.cj.jdbc.StatementImpl@5e5d171f  
Row is impacted: 1  
ID: 1, Age: 30, First Name: Saim, Lastname: Ayhan  
ID: 2, Age: 32, First Name: Altuğ, Lastname: Paylan  
ID: 3, Age: 34, First Name: Nuri, Lastname: Zor  
ID: 4, Age: 40, First Name: Alper, Lastname: Tunga  
ID: 5, Age: 21, First Name: Boğaç, Lastname: Han  
ID: 6, Age: 48, First Name: Orçun, Lastname: Zaloğlu  
ID: 7, Age: 59, First Name: Efkan, Lastname: Özür  
ID: 8, Age: 50, First Name: Süleyman, Lastname: İbret  
ID: 9, Age: 29, First Name: Erman, Lastname: Hantallı  
ID: 10, Age: 39, First Name: Ersan, Lastname: Şamar  
ID: 11, Age: 47, First Name: Buket, Lastname: Kal
```

## Closing Statement Object

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

A simple call to the `close()` method will do the job. If you close the Connection object first, it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

```
Statement stmt = null;
try {
    stmt = conn.createStatement();
    . .
}
catch (SQLException e) {
    . .
}
finally {
    stmt.close();
}
```

For a better understanding, we suggest you to study the [Statement - Example tutorial](#).

## The PreparedStatement Objects

The `PreparedStatement` interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

## Creating PreparedStatement Object

```
PreparedStatement pstmt = null;
try {
    String SQL = "Update Employees SET age = ? WHERE id = ?";
    pstmt = conn.prepareStatement(SQL);
    . .
}
catch (SQLException e) {
    . .
}
finally {
    . .
}
```

All parameters in JDBC are represented by the `?` symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The `setXXX()` methods bind values to the parameters, where `XXX` represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an `SQLException`.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

All of the `Statement` object's methods for interacting with the database (a) `execute()`, (b) `executeQuery()`, and (c) `executeUpdate()` also work with the `PreparedStatement` object. However, the methods are modified to use SQL statements that can input the parameters.

## Closing PreparedStatement Object

Just as you close a Statement object, for the same reason you should also close the PreparedStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

```
PreparedStatement pstmt = null;
try {
    String SQL = "Update Employees SET age = ? WHERE id = ?";
    pstmt = conn.prepareStatement(SQL);
    . . .
}
catch (SQLException e) {
    . . .
}
finally {
    pstmt.close();
}
```

For a better understanding, let us study [Prepare - Example Code](#).

## The CallableStatement Objects

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

## Creating CallableStatement Object

Suppose, you need to execute the following Oracle stored procedure –

```
CREATE OR REPLACE PROCEDURE getEmpName
    (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS
BEGIN
    SELECT first INTO EMP_FIRST
    FROM Employees
    WHERE ID = EMP_ID;
END;
```

**NOTE** – Above stored procedure has been written for Oracle, but we are working with MySQL database so, let us write same stored procedure for MySQL as follows to create it in EMP database –

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$

CREATE PROCEDURE `EMP`.`getEmpName`
    (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
    SELECT first INTO EMP_FIRST
    FROM Employees
    WHERE ID = EMP_ID;
END $$

DELIMITER ;
```

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

Here are the definitions of each –

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

The following code snippet shows how to employ the **Connection.prepareCall()** method to instantiate a **CallableStatement** object based on the preceding stored procedure

```
CallableStatement cstmt = null;
try {
    String SQL = "{call getEmpName (?, ?)}";
    cstmt = conn.prepareCall (SQL);
    ...
}
catch (SQLException e) {
    ...
}
finally {
    ...
}
```

The String variable SQL, represents the stored procedure, with parameter placeholders.

Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException.

If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return.

Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

## Closing CallableStatement Object

Just as you close other Statement object, for the same reason you should also close the CallableStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the CallableStatement object. You can explicitly close the CallableStatement object to ensure proper cleanup.

```
CallableStatement cstmt = null;
try {
    String SQL = "{call getEmpName (?, ?)}";
    cstmt = conn.prepareCall (SQL);
    . . .
}
catch (SQLException e) {
    . . .
}
finally {
    cstmt.close();
}
```

For a better understanding, I would suggest to study [Callable - Example Code](#).

# JDBC - Result Sets

The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The `java.sql.ResultSet` interface represents the result set of a database query.

A `ResultSet` object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a `ResultSet` object.

The methods of the `ResultSet` interface can be broken down into three categories –

- **Navigational methods** – Used to move the cursor around.
- **Get methods** – Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods** – Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the `ResultSet`. These properties are designated when the corresponding `Statement` that generates the `ResultSet` is created.

JDBC provides the following connection methods to create statements with desired `ResultSet` –

- `createStatement(int RSType, int RSConcurrency);`
- `prepareStatement(String SQL, int RSType, int RSConcurrency);`
- `prepareCall(String sql, int RSType, int RSConcurrency);`

The first argument indicates the type of a `ResultSet` object and the second argument is one of two `ResultSet` constants for specifying whether a result set is read-only or updatable.

## Type of ResultSet

The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE\_FORWARD\_ONLY.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

## Concurrency of ResultSet

The possible RSConcurrency are given below. If you do not specify any Concurrency type, you will automatically get one that is CONCUR\_READ\_ONLY.

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

All our examples written so far can be written as follows, which initializes a Statement object to create a forward-only, read only ResultSet object –

```
try {
    Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);
}
catch(Exception ex) {
    ....
}
finally {
    ....
}
```

## Navigating a Result Set

There are several methods in the ResultSet interface that involve moving the cursor, including –

S.N.	Methods & Description
1	<b>public void beforeFirst() throws SQLException</b> Moves the cursor just before the first row.
2	<b>public void afterLast() throws SQLException</b> Moves the cursor just after the last row.
3	<b>public boolean first() throws SQLException</b> Moves the cursor to the first row.
4	<b>public void last() throws SQLException</b> Moves the cursor to the last row.
5	<b>public boolean absolute(int row) throws SQLException</b> Moves the cursor to the specified row.
6	<b>public boolean relative(int row) throws SQLException</b> Moves the cursor the given number of rows forward or backward, from where it is currently pointing.
7	<b>public boolean previous() throws SQLException</b> Moves the cursor to the previous row. This method returns false if the previous row is off the result set.
8	<b>public boolean next() throws SQLException</b> Moves the cursor to the next row. This method returns false if there are no more rows in the result set.
9	<b>public int getRow() throws SQLException</b> Returns the row number that the cursor is pointing to.
10	<b>public void moveToInsertRow() throws SQLException</b> Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered.
11	<b>public void moveToCurrentRow() throws SQLException</b> Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing

## Viewing a Result Set

The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions –

- One that takes in a column name.
- One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet –

S.N.	Methods & Description
1	<code>public int getInt(String columnName) throws SQLException</code> Returns the int in the current row in the column named columnName.
2	<code>public int getInt(int columnIndex) throws SQLException</code> Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Similarly, there are get methods in the ResultSet interface for each of the eight Java primitive types, as well as common types such as java.lang.String, java.lang.Object, and java.net.URL.

There are also methods for getting SQL data types java.sql.Date, java.sql.Time, java.sql.Timestamp, java.sql.Clob, and java.sql.Blob. Check the documentation for more information about using these SQL data types.

For a better understanding, let us study [Viewing - Example Code](#).

## Updating a Result Set

The ResultSet interface contains a collection of update methods for updating the data of a result set.

As with the get methods, there are two update methods for each data type –

- One that takes in a column name.
- One that takes in a column index.

For example, to update a String column of the current row of a result set, you would use one of the following updateString() methods –

S.N.	Methods & Description
1	<b>public void updateString(int columnIndex, String s) throws SQLException</b> Changes the String in the specified column to the value of s.
2	<b>public void updateString(String columnName, String s) throws SQLException</b> Similar to the previous method, except that the column is specified by its name instead of its index.

There are update methods for the eight primitive data types, as well as String, Object, URL, and the SQL data types in the java.sql package.

Updating a row in the result set changes the columns of the current row in the ResultSet object, but not in the underlying database. To update your changes to the row in the database, you need to invoke one of the following methods.

S.N.	Methods & Description
1	<b>public void updateRow()</b> Updates the current row by updating the corresponding row in the database.
2	<b>public void deleteRow()</b> Deletes the current row from the database
3	<b>public void refreshRow()</b> Refreshes the data in the result set to reflect any recent changes in the database.
4	<b>public void cancelRowUpdates()</b> Cancels any updates made on the current row.
5	<b>public void insertRow()</b> Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.

For a better understanding, let us study the [Updating - Example Code](#).

# 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[ ]	getBytes	getBytes
BINARY	byte[ ]	getBytes	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
```

Now let us compile the above example as follows –

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

When you run `JDBCExample`, it produces the following result –

```
C:\>java SqlDateTime
The Java Date is:Tue Aug 18 13:46:02 GMT+04:00 2009
The SQL DATE is: 2009-08-18
The SQL TIME is: 13:46:02
The SQL TIMESTAMP is: 2009-08-18 13:46:02.828
C:\>
```

## Handling NULL Values

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;
}
```

# JDBC - Transactions

If your JDBC Connection is in *auto-commit* mode, which it is by default, then every SQL statement is committed to the database upon its completion.

That may be fine for simple applications, but there are three reasons why you may want to turn off the auto-commit and manage your own transactions –

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

Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.

To enable manual- transaction support instead of the *auto-commit* mode that the JDBC driver uses by default, use the Connection object's **setAutoCommit()** method. If you pass a boolean false to **setAutoCommit( )**, you turn off auto-commit. You can pass a boolean true to turn it back on again.

For example, if you have a Connection object named conn, code the following to turn off auto-commit –

```
conn.setAutoCommit(false);
```

## Commit & Rollback

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

```
conn.commit();
```

Otherwise, to roll back updates to the database made using the Connection named conn, use the following code –

```
conn.rollback();
```

The following example illustrates the use of a commit and rollback object –

```
try{  
    //Assume a valid connection object conn  
    conn.setAutoCommit(false);  
    Statement stmt = conn.createStatement();  
  
    String SQL = "INSERT INTO Employees " +  
                "VALUES (106, 20, 'Rita', 'Tez')";  
    stmt.executeUpdate(SQL);  
    //Submit a malformed SQL statement that breaks  
    String SQL = "INSERTED IN Employees " +  
                "VALUES (107, 22, 'Sita', 'Singh')";  
    stmt.executeUpdate(SQL);  
    // If there is no error.  
    conn.commit();  
}catch(SQLException se){  
    // If there is any error.  
    conn.rollback();  
}
```

In this case, none of the above INSERT statement would success and everything would be rolled back.

## Using Savepoints

The new JDBC 3.0 Savepoint interface gives you the additional transactional control. Most modern DBMS, support savepoints within their environments such as Oracle's PL/SQL.

When you set a savepoint you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

The Connection object has two new methods that help you manage savepoints –

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

There is one **rollback (String savepointName)** method, which rolls back work to the specified savepoint.

The following example illustrates the use of a Savepoint object –

```
try{  
    //Assume a valid connection object conn  
    conn.setAutoCommit(false);  
    Statement stmt = conn.createStatement();  
  
    //set a Savepoint  
    Savepoint savepoint1 = conn.setSavepoint("Savepoint1");  
    String SQL = "INSERT INTO Employees " +  
        "VALUES (106, 20, 'Rita', 'Tez')";  
    stmt.executeUpdate(SQL);  
    //Submit a malformed SQL statement that breaks  
    String SQL = "INSERTED IN Employees " +  
        "VALUES (107, 22, 'Sita', 'Tez')";  
    stmt.executeUpdate(SQL);  
    // If there is no error, commit the changes.  
    conn.commit();  
  
}catch(SQLException se){  
    // If there is any error.  
    conn.rollback(savepoint1);  
}
```

In this case, none of the above INSERT statement would success and everything would be rolled back.

# JDBC - Exceptions Handling

Exception handling allows you to handle exceptional conditions such as program-defined errors in a controlled fashion.

When an exception condition occurs, an exception is thrown. The term thrown means that current program execution stops, and the control is redirected to the nearest applicable catch clause. If no applicable catch clause exists, then the program's execution ends.

JDBC Exception handling is very similar to the Java Exception handling but for JDBC, the most common exception you'll deal with is [java.sql.SQLException](#).

## SQLException Methods

An SQLException can occur both in the driver and the database. When such an exception occurs, an object of type SQLException will be passed to the catch clause.

The passed SQLException object has the following methods available for retrieving additional information about the exception –

Method	Description
getErrorCode()	Gets the error number associated with the exception.
getMessage()	Gets the JDBC driver's error message for an error, handled by the driver or gets the Oracle error number and message for a database error.
getSQLState()	Gets the XOPEN SQLstate string. For a JDBC driver error, no useful information is returned from this method. For a database error, the five-digit XOPEN SQLstate code is returned. This method can return null.
getNextException()	Gets the next Exception object in the exception chain.
printStackTrace( )	Prints the current exception, or throwable, and it's backtrace to a standard error stream.
printStackTrace(PrintStream s)	Prints this throwable and its backtrace to the print stream you specify.
printStackTrace(PrintWriter w)	Prints this throwable and it's backtrace to the print writer you specify.

By utilizing the information available from the Exception object, you can catch an exception and continue your program appropriately. Here is the general form of a try block –

```
try {
    // Your risky code goes between these curly braces!!!
}
catch(Exception ex) {
    // Your exception handling code goes between these
    // curly braces, similar to the exception clause
    // in a PL/SQL block.
}
finally {
    // Your must-always-be-executed code goes between these
    // curly braces. Like closing database connection.
}
```

## Example

Study the following example code to understand the usage of **try....catch....finally** blocks.

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class JDBCExample {
    static final String DB_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";
    static final String USER = "guest";
    static final String PASS = "guest123";
    static final String QUERY = "{call getEmpName (?, ?)}";

    public static void main(String[] args) {
        // Open a connection
        try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
            CallableStatement stmt = conn.prepareCall(QUERY);
        ) {
            // Bind values into the parameters.
            stmt.setInt(1, 1); // This would set ID
            // Because second parameter is OUT so register it
            stmt.registerOutParameter(2, java.sql.Types.VARCHAR);
            //Use execute method to run stored procedure.
            System.out.println("Executing stored procedure...");
            stmt.execute();
            //Retrieve employee name with getXXX method
            String empName = stmt.getString(2);
            System.out.println("Emp Name with ID: 1 is " + empName);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Now, let us compile the above example as follows –

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

When you run **JDBCExample**, it produces the following result if there is no problem, otherwise the corresponding error would be displayed –

```
C:\>java JDBCExample
Executing stored procedure...
Emp Name with ID: 1 is Zara
C:\>
```

Try the above example by passing wrong database name or wrong username or password and check the result.

# JDBC - Batch Processing

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

When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.

- JDBC drivers are not required to support this feature. You should use the `DatabaseMetaData.supportsBatchUpdates()` method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.
- The `addBatch()` method of `Statement`, `PreparedStatement`, and `CallableStatement` is used to add individual statements to the batch. The `executeBatch()` is used to start the execution of all the statements grouped together.
- The `executeBatch()` returns an array of integers, and each element of the array represents the update count for the respective update statement.
- Just as you can add statements to a batch for processing, you can remove them with the `clearBatch()` method. This method removes all the statements you added with the `addBatch()` method. However, you cannot selectively choose which statement to remove.

## Batching with Statement Object

Here is a typical sequence of steps to use Batch Processing with Statement Object –

- Create a Statement object using either `createStatement()` methods.
- Set auto-commit to false using `setAutoCommit()`.
- Add as many as SQL statements you like into batch using `addBatch()` method on created statement object.
- Execute all the SQL statements using `executeBatch()` method on created statement object.
- Finally, commit all the changes using `commit()` method.

## Example

The following code snippet provides an example of a batch update using Statement object –

```
// Create statement object
Statement stmt = conn.createStatement();

// Set auto-commit to false
conn.setAutoCommit(false);

// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
    "VALUES(200,'Zia', 'Ali', 30)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
    "VALUES(201,'Raj', 'Kumar', 35)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "UPDATE Employees SET age = 35 " +
    "WHERE id = 100";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create an int[] to hold returned values
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```

## Batching with PreparedStatement Object

Here is a typical sequence of steps to use Batch Processing with PreparedStatement Object –

1. Create SQL statements with placeholders.
2. Create PreparedStatement object using either *prepareStatement()* methods.
3. Set auto-commit to false using *setAutoCommit()*.
4. Add as many as SQL statements you like into batch using *addBatch()* method on created statement object.
5. Execute all the SQL statements using *executeBatch()* method on created statement object.
6. Finally, commit all the changes using *commit()* method.

The following code snippet provides an example of a batch update using PreparedStatement object –

```
// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
    "VALUES(?, ?, ?, ?);"

// Create PreparedStatement object
PreparedStatement pstmt = conn.prepareStatement(SQL);

//Set auto-commit to false
conn.setAutoCommit(false);

// Set the variables
pstmt.setInt( 1, 400 );
pstmt.setString( 2, "Pappu" );
pstmt.setString( 3, "Singh" );
pstmt.setInt( 4, 33 );
// Add it to the batch
pstmt.addBatch();

// Set the variables
pstmt.setInt( 1, 401 );
pstmt.setString( 2, "Pawan" );
pstmt.setString( 3, "Singh" );
pstmt.setInt( 4, 31 );
// Add it to the batch
pstmt.addBatch();

//add more batches
.
.
.

//Create an int[] to hold returned values
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```

# JDBC - Stored Procedure

We have learnt how to use **Stored Procedures** in JDBC while discussing the [JDBC - Statements](#) chapter. This chapter is similar to that section, but it would give you additional information about JDBC SQL escape syntax.

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

## Creating CallableStatement Object

Suppose, you need to execute the following Oracle stored procedure –

```
CREATE OR REPLACE PROCEDURE getEmpName
    (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS
BEGIN
    SELECT first INTO EMP_FIRST
    FROM Employees
    WHERE ID = EMP_ID;
END;
```

**NOTE** – Above stored procedure has been written for Oracle, but we are working with MySQL database so, let us write same stored procedure for MySQL as follows to create it in EMP database.

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$

CREATE PROCEDURE `EMP`.`getEmpName`
    (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
    SELECT first INTO EMP_FIRST
    FROM Employees
    WHERE ID = EMP_ID;
END $$

DELIMITER ;
```

Three types of parameters exist – IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

Here are the definitions of each –

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

The following code snippet shows how to employ the `Connection.prepareCall()` method to instantiate a `CallableStatement` object based on the preceding stored procedure

```
-  
  
CallableStatement cstmt = null;  
try {  
    String SQL = "{call getEmpName (?, ?)}";  
    cstmt = conn.prepareCall (SQL);  
    . . .  
}  
catch (SQLException e) {  
    . . .  
}  
finally {  
    . . .  
}
```

The String variable SQL represents the stored procedure, with parameter placeholders.

Using `CallableStatement` objects is much like using `PreparedStatement` objects. You must bind values to all the parameters before executing the statement, or you will receive an `SQLException`.

If you have IN parameters, just follow the same rules and techniques that apply to a `PreparedStatement` object; use the `setXXX()` method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters, you must employ an additional `CallableStatement` method, `registerOutParameter()`. The `registerOutParameter()` method binds the JDBC data type to the data type the stored procedure is expected to return.

Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate `getXXX()` method. This method casts the retrieved value of SQL type to a Java data type.

## Closing CallableStatement Object

Just as you close other Statement object, for the same reason you should also close the `CallableStatement` object.

A simple call to the `close()` method will do the job. If you close the `Connection` object first, it will close the `CallableStatement` object as well. However, you should always explicitly close the `CallableStatement` object to ensure proper cleanup.

```
CallableStatement cstmt = null;  
try {  
    String SQL = "{call getEmpName (?, ?)}";  
    cstmt = conn.prepareCall (SQL);  
    . . .  
}  
catch (SQLException e) {  
    . . .  
}  
finally {  
    cstmt.close();  
}
```

## JDBC SQL Escape Syntax

The escape syntax gives you the flexibility to use database specific features unavailable to you by using standard JDBC methods and properties.

The general SQL escape syntax format is as follows –

```
{keyword 'parameters'}
```

Here are the following escape sequences, which you would find very useful while performing the JDBC programming –

### d, t, ts Keywords

They help identify date, time, and timestamp literals. As you know, no two DBMSs represent time and date the same way. This escape syntax tells the driver to render the date or time in the target database's format. For Example –

```
{d 'yyyy-mm-dd'}
```

Where yyyy = year, mm = month; dd = date. Using this syntax {d '2009-09-03'} is March 9, 2009.

Here is a simple example showing how to INSERT date in a table –

```
//Create a Statement object
stmt = conn.createStatement();
//Insert data ==> ID, First Name, Last Name, DOB
String sql="INSERT INTO STUDENTS VALUES" +
    "(100,'Zara','Ali', {d '2001-12-16'})";

stmt.executeUpdate(sql);
```

Similarly, you can use one of the following two syntaxes, either **t** or **ts** –

```
{t 'hh:mm:ss'}
```

Where hh = hour; mm = minute; ss = second. Using this syntax {t '13:30:29'} is 1:30:29 PM.

```
{ts 'yyyy-mm-dd hh:mm:ss'}
```

This is combined syntax of the above two syntax for 'd' and 't' to represent timestamp.

## escape Keyword

This keyword identifies the escape character used in LIKE clauses. Useful when using the SQL wildcard %, which matches zero or more characters. For example –

```
String sql = "SELECT symbol FROM MathSymbols  
    WHERE symbol LIKE '\%' {escape '\\'}";  
stmt.execute(sql);
```

If you use the backslash character (\) as the escape character, you also have to use two backslash characters in your Java String literal, because the backslash is also a Java escape character.

## fn Keyword

This keyword represents scalar functions used in a DBMS. For example, you can use SQL function *length* to get the length of a string –

```
{fn length('Hello World')}
```

This returns 11, the length of the character string 'Hello World'.

## call Keyword

This keyword is used to call the stored procedures. For example, for a stored procedure requiring an IN parameter, use the following syntax –

```
{call my_procedure(?)};
```

For a stored procedure requiring an IN parameter and returning an OUT parameter, use the following syntax –

```
{? = call my_procedure(?)};
```

## oj Keyword

This keyword is used to signify outer joins. The syntax is as follows –

```
{oj outer-join}
```

Where outer-join = table {LEFT|RIGHT|FULL} OUTERJOIN {table | outer-join} on search-condition. For example –

```
String sql = "SELECT Employees  
    FROM {oj ThisTable RIGHT  
        OUTER JOIN ThatTable on id = '100'}";  
stmt.execute(sql);
```

# JDBC - Streaming ASCII and Binary Data

A PreparedStatement object has the ability to use input and output streams to supply parameter data. This enables you to place entire files into database columns that can hold large values, such as CLOB and BLOB data types.

There are following methods, which can be used to stream data –

- **setAsciiStream()** – This method is used to supply large ASCII values.
- **setCharacterStream()** – This method is used to supply large UNICODE values.
- **setBinaryStream()** – This method is used to supply large binary values.

The setXXXStream() method requires an extra parameter, the file size, besides the parameter placeholder. This parameter informs the driver how much data should be sent to the database using the stream.

```

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestApplication {
    static final String DB_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";
    static final String USER = "guest";
    static final String PASS = "guest123";
    static final String QUERY = "SELECT Data FROM XML_Data WHERE id=100";
    static final String INSERT_QUERY="INSERT INTO XML_Data VALUES (?,?)";
    static final String CREATE_TABLE_QUERY = "CREATE TABLE XML_Data (id INTEGER, Data LONG)";
    static final String DROP_TABLE_QUERY = "DROP TABLE XML_Data";
    static final String XML_DATA = "<Employee><id>100</id><first>Zara</first><last>Ali</last><Salary>10000</Salary><Dob>18-08-1978</Dob></Employee>"

    public static void createXMLTable(Statement stmt)
        throws SQLException{
        System.out.println("Creating XML_Data table..." );
        //Drop table first if it exists.
        try{
            stmt.executeUpdate(DROP_TABLE_QUERY);
        }catch(SQLException se){
        }
        stmt.executeUpdate(CREATE_TABLE_QUERY);
    }
}

```

```

public static void main(String[] args) {
    // Open a connection
    try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
        Statement stmt = conn.createStatement();
        PreparedStatement pstmt = conn.prepareStatement(INSERT_QUERY);
    ) {
        createXMLTable(stmt);

        ByteArrayInputStream bis = new ByteArrayInputStream(XML_DATA.getBytes());

        pstmt.setInt(1,100);
        pstmt.setAsciiStream(2,bis,XML_DATA.getBytes().length);
        pstmt.execute();

        //Close input stream
        bis.close();

        ResultSet rs = stmt.executeQuery(QUERY);
        // Get the first row
        if (rs.next()){
            //Retrieve data from input stream
            InputStream xmlInputStream = rs.getAsciiStream (1);
            int c;
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            while (( c = xmlInputStream.read ()) != -1)
                bos.write(c);
            //Print results
            System.out.println(bos.toString());
        }
        // Clean-up environment
        rs.close();

    } catch (SQLException | IOException e) {
        e.printStackTrace();
    }
}

```

Now let us compile the above example as follows –

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

When you run **FirstApplication**, it produces the following result –

```
C:\>java FirstApplication
Creating XML_Data table...
<Employee><id>100</id><first>Zara</first><last>Ali</last><Salary>10000</Salary><Dob>18-08-1978</Dob></Employee>
C:\>
```

# JDBC - Create Database Example

This tutorial provides an example on how to create a Database using JDBC application. Before executing the following example, make sure you have the following in place –

- You should have admin privilege to create a database in the given schema. To execute the following example, you need to replace the *username* and *password* with your actual user name and password.
- Your MySQL or whatever database is up and running.

## Required Steps

The following steps are required to create a new Database using 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.
- **Open a connection** – Requires using the `DriverManager.getConnection()` method to create a Connection object, which represents a physical connection with the database server.

To create a new database, you need not give any database name while preparing database URL as mentioned in the below example.

- **Execute a query** – Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Clean up the environment** . `try with resources` automatically closes the resources.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

new *
> public class CH01_CreatingDatabase {
    new *
>     public static void main(String[] args) {
        String url="jdbc:mysql://localhost/";
        String userName = "root";
        String password = "1234";

        try(Connection connection = DriverManager.getConnection(url, userName, password);
            Statement statement = connection.createStatement();) {
            String sql = "CREATE DATABASE STUDENTS";
            statement.executeUpdate(sql);
            System.out.println("Database created successfully...");

        }catch (SQLException e){
            e.printStackTrace();
        }
    }
}
```

Database created successfully...

Process finished with exit code 0

## JDBC - Select Database Example

This chapter provides an example on how to select a Database using JDBC application. Before executing the following example, make sure you have the following in place –

- To execute the following example you need to replace the *username* and *password* with your actual user name and password.
- Your MySQL or whatever database you are using, is up and running.

## Required Steps

The following steps are required to create a new Database using JDBC application –

- **Import the packages** – Requires that you include the packages containing the JDBC classes needed for the database programming. Most often, using `import java.sql.*` will suffice.
- **Open a connection** – Requires using the `DriverManager.getConnection()` method to create a Connection object, which represents a physical connection with a **selected** database.

Selection of database is made while you prepare database URL. Following example would make connection with **STUDENTS** database.

- **Clean up the environment** – try with resources automatically closes the resources.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

new *

public class CH03_SelectDatabase {
    new *
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/newbase11";
        String userName = "root";
        String password = "1234";
        try (Connection connection = DriverManager.getConnection(url, userName, password);
            Statement statement = connection.createStatement();) {
            System.out.println("Database bağlandı.....");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Database bağlandı.....

Process finished with exit code 0

# JDBC - Drop Database Example

This chapter provides an example on how to drop an existing Database using JDBC application. Before executing the following example, make sure you have the following in place –

- To execute the following example you need to replace the *username* and *password* with your actual user name and password.
- Your MySQL is up and running.

**NOTE:** This is a serious operation and you have to make a firm decision before proceeding to delete a database because everything you have in your database would be lost.

## Required Steps

The following steps are required to create a new Database using 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.
- **Open a connection** – Requires using the `DriverManager.getConnection()` method to create a Connection object, which represents a physical connection with a database server.

Deleting a database does not require database name to be in your database URL. Following example would delete **STUDENTS** database.

- **Execute a query** – Requires using an object of type Statement for building and submitting an SQL statement to delete the database.
- **Clean up the environment** – try with resources automatically closes the resources.

```
✓ import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

new *
> public class CH01_DropDatabase {
    new *
>     public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/";
        String username = "root";
        String password = "1234";
        try(Connection connection = DriverManager.getConnection(url, username, password);
            Statement statement = connection.createStatement()){
            statement.executeUpdate( sql: "DROP DATABASE EMPLOYEES");
            System.out.println("Database EMPLOYEES silindi....");
        }catch (SQLException e){
            e.printStackTrace();
        }
    }
}
```

Database teacher silindi....

Process finished with exit code 0

# JDBC - Create Table Example

This chapter provides an example on how to create a table using JDBC application. Before executing the following example, make sure you have the following in place –

- To execute the following example you can replace the *username* and *password* with your actual user name and password.
- Your MySQL is up and running.

## Required Steps

The following steps are required to create a new Database using 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.
- **Open a connection** – Requires using the `DriverManager.getConnection()` method to create a Connection object, which represents a physical connection with a database server.
- **Execute a query** – Requires using an object of type Statement for building and submitting an SQL statement to create a table in a selected database.
- **Clean up the environment** – try with resources automatically closes the resources.

```
1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.SQLException;
4 import java.sql.Statement;
5
6 new *
7 D public class CreatingTable03 {
8     new *
9     public static void main(String[] args) {
10         String url = "jdbc:mysql://localhost:3306/newbase22";
11         String username = "root";
12         String password = "1234";
13
14         try (Connection connection = DriverManager.getConnection(url, username, password);
15              Statement statement = connection.createStatement();) {
16             statement.executeUpdate(
17                 sql: "CREATE TABLE TABLO03(" +
18                     "id int primary key auto_increment" +
19                     "name varchar(50)," +
20                     "maas int);");
21
22             System.out.println("TABLO03 TABLOSU OLUSTURULDU");
23         } catch (SQLException e) {
24             e.printStackTrace();
25         }
26     }
27 }
```

Run CreatingTable03

C:\Users\aronik\.jdks\openjdk-20.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.1\lib\javassist-agent.jar" -Dfile.encoding=UTF-8 TABLO03 TABLOSU OLUSTURULDU

Process finished with exit code 0

# JDBC - Drop Table Example

This chapter provides an example on how to delete a table using JDBC application. Before executing the following example, make sure you have the following in place –

- To execute the following example you can replace the *username* and *password* with your actual user name and password.
- Your MySQL or whatever database you are using, is up and running.

**NOTE** Reformatting JDBC Tutorial This is a serious operation and you have to make a firm decision before proceeding to delete a table, because everything you have in your table would be lost.

## Required Steps

The following steps are required to create a new Database using 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.
- **Open a connection** – Requires using the `DriverManager.getConnection()` method to create a Connection object, which represents a physical connection with a database server.
- **Execute a query**Reformatting JDBC Tutorial Requires using an object of type Statement for building and submitting an SQL statement to drop a table in a selected database.
- **Clean up the environment** Reformatting JDBC Tutorial try with resources automatically closes the resources.

```
1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.SQLException;
4 import java.sql.Statement;
5
6 new *
7 public class DropTable01 {
8     new *
9     public static void main(String[] args) {
10         String url = "jdbc:mysql://localhost:3306/newbase22";
11         String username = "root";
12         String password = "1234";
13
14         try(Connection connection = DriverManager.getConnection(url, username, password);
15             Statement statement = connection.createStatement();){
16             statement.executeUpdate( sql: "DROP TABLE TABLO1");
17             System.out.println("TABLO1 SİLİNDİ");
18         }catch (SQLException e){
19             e.printStackTrace();
20         }
21     }
22 }
```

Run DropTable01 ×

C:\Users\aronik\.jdks\openjdk-20.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.1\lib\idea\_rt.jar" -Dfile.encoding=UTF-8

TABLO1 SİLİNDİ

Process finished with exit code 0

# JDBC - Insert Records Example



This chapter provides an example on how to insert records in a table using JDBC application. Before executing following example, make sure you have the following in place –

- To execute the following example you can replace the *username* and *password* with your actual user name and password.
- Your MySQL or whatever database you are using is up and running.

## Required Steps

The following steps are required to create a new Database using 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 communications channel with the database.
- **Open a connection** – Requires using the `DriverManager.getConnection()` method to create a Connection object, which represents a physical connection with a database server.
- **Execute a query** – Requires using an object of type Statement for building and submitting an SQL statement to insert records into a table.
- **Clean up the environment** try with resources automatically closes the resources.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

new *
> public class InsertInto02 {
    new *
>     public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/teachers";
        String userName = "root";
        String password = "1234";

        try (Connection connection = DriverManager.getConnection(url, userName, password);
            Statement statement = connection.createStatement()) {
            statement.executeUpdate(sql: "INSERT INTO tablo01 (name, maas) values ('Ersan Sıço', 2200), ('Erman Toro', 2300), ('Süleyman Cico', 4500)");
            System.out.println("TABLO01 E VERİ GİRİŞİ YAPILDI");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```
InsertInto02 x
| @ : 
C:\Users\aronik\.jdks\openjdk-20.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.1.3.A\lib\idea_rt.jar=62646:C:\Program Files\TABLO01 E VERİ GİRİŞİ YAPILDI

Process finished with exit code 0
```

	id	name	maas
▶	1	Ersan Sıço	2200
	2	Erman Toro	2300
	3	Süleyman Cico	4500
*	NULL	NULL	NULL

# JDBC - Select Records Example

This chapter provides an example on how to select/ fetch records from a table using JDBC application. Before executing the following example, make sure you have the following in place -

- To execute the following example you can replace the username and password with your actual user name and password.
- Your MySQL or whatever database you are using is up and running.

## Required Steps

The following steps are required to create a new Database using 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.
- **Open a connection** – Requires using the `DriverManager.getConnection()` method to create a `Connection` object, which represents a physical connection with a database server.
- **Execute a query** – Requires using an object of type `Statement` for building and submitting an SQL statement to select (i.e. fetch ) records from a table.
- **Extract Data** – Once SQL query is executed, you can fetch records from the table.
- **Clean up the environment** – try with resources automatically closes the resources.

```
import java.sql.*;

new *
public class SelectRecordExamples01 {
    new *
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:/newbase24";
        String userName = "root";
        String password = "1234";
        String query = "SELECT name, age FROM employees";
        try(Connection connection = DriverManager.getConnection(url, userName, password);
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery(query)){
            while (resultSet.next()){
                System.out.println("İsim: " + resultSet.getString( columnLabel: "name"));
                System.out.println("Yaşı: " + resultSet.getInt( columnLabel: "age"));
                System.out.println("*****");
            }
        }catch (SQLException e){
            e.printStackTrace();
        }
    }
}
```

```
İsim: Saim Ayhan
Yaşı: 30
*****
İsim: Nur Peko
Yaşı: 28
*****
İsim: Altuğ Paylan
Yaşı: 32
*****
İsim: Nuri Zor
Yaşı: 34
*****
İsim: Alper Tunga
Yaşı: 40
*****
İsim: Boğaç Han
Yaşı: 45
*****
İsim: Orçun Zaaloğlu
Yaşı: 48
*****
İsim: Efkan Özür
Yaşı: 59
*****
İsim: Süleyman İbret
Yaşı: 50
*****
İsim: Erman Hantallı
Yaşı: 29
*****
İsim: Ersan Şamar
Yaşı: 39
*****
İsim: Buket Kal
Yaşı: 47
*****
```

# JDBC - Delete Records Example

This chapter provides an example on how to delete records from a table using JDBC application. Before executing following example, make sure you have the following in place -

- To execute the following example you can replace the username and password with your actual user name and password.
- Your MySQL or whatever database you are using is up and running.

## Required Steps

The following steps are required to create a new Database using 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 communications channel with the database.
- **Open a connection** – Requires using the `DriverManager.getConnection()` method to create a Connection object, which represents a physical connection with a database server.
- **Execute a query** – Requires using an object of type Statement for building and submitting an SQL statement to delete records from a table. This Query makes use of the **WHERE** clause to delete conditional records.
- **Clean up the environment** – try with resources automatically closes the resources.

```

import java.sql.*;

AYTEKİN KAPLAN *
public class DeleteRecords01 {
    AYTEKİN KAPLAN *
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/newbase23";
        String user = "root";
        String password = "1234";
        String query = "SELECT id, name, age FROM mytable04";
        try (Connection connection = DriverManager.getConnection(url, user, password);
            Statement statement = connection.createStatement();) {
            statement.executeUpdate( sql: "DELETE FROM mytable04 where id=3");
            System.out.println("DATA TABLO MYTABLE01 DEN SİSLİNDİ");
            ResultSet resultSet = statement.executeQuery(query);
            while (resultSet.next()) {
                System.out.println("ID: " + resultSet.getInt(columnLabel: "id"));
                System.out.println("NAME : " + resultSet.getString(columnLabel: "name"));
                System.out.println("AGE: " + resultSet.getByte(columnLabel: "age"));
                System.out.println("-----");
            }
            System.out.println("TABLONUN SON ŞEKLİ BUDUR");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

	<b>id</b>	<b>name</b>	<b>age</b>
1	Alice	25	
2	Bob	30	
3	Charlie	22	
4	David	28	
5	Eve	35	
	HULL	HULL	HULL

**Result Grid** | Filter

	<b>id</b>	<b>name</b>	<b>age</b>
1	Alice	25	
2	Bob	30	
4	David	28	
5	Eve	35	
	HULL	HULL	HULL

# JDBC - WHERE Clause Example

This chapter provides an example on how to select records from a table using JDBC application. This would add additional conditions using WHERE clause while selecting records from the table. Before executing the following example, make sure you have the following in place -

- To execute the following example you can replace the username and password with your actual user name and password.
- Your MySQL or whatever database you are using, is up and running.

## Required Steps

The following steps are required to create a new Database using JDBC application -

- **Import the packages** – Requires that you include the packages containing the JDBC classes needed for the 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 communications channel with the database.
- **Open a connection** – Requires using the DriverManager.getConnection() method to create a Connection object, which represents a physical connection with a database server.
- **Execute a query** – Requires using an object of type Statement for building and submitting an SQL statement to fetch records from a table, which meet the given condition. This Query makes use of the **WHERE** clause to select records.
- **Clean up the environment** – try with resources automatically closes the resources.

```
import java.sql.*;

new *
D  public class WhereClauseUsing01 {
    new *
D  public static void main(String[] args) {
    String url = "jdbc:mysql://localhost/newbase23";
    String user = "root";
    String password = "1234";
    String query = "SELECT id, name, age FROM mytable05";
    try (Connection connection = DriverManager.getConnection(url, user, password);
        Statement statement = connection.createStatement();) {
        ResultSet resultSet = statement.executeQuery(query);
        System.out.println("KOŞUL BELİRTİLMEÐEN");
        while (resultSet.next()) {
            System.out.println("ID: " + resultSet.getInt(columnLabel: "id"));
            System.out.println("NAME : " + resultSet.getString(columnLabel: "name"));
            System.out.println("AGE: " + resultSet.getByte(columnLabel: "age"));
            System.out.println("-----");
        }
        System.out.println("KOŞUL BELİRTİLDİKTEM SONRA");
        query = "SELECT id, name, age FROM mytable05 where age<=25";
        resultSet = statement.executeQuery(query);
        while (resultSet.next()) {
            System.out.println("ID: " + resultSet.getInt(columnLabel: "id"));
            System.out.println("NAME : " + resultSet.getString(columnLabel: "name"));
            System.out.println("AGE: " + resultSet.getByte(columnLabel: "age"));
            System.out.println("-----");
        }
        resultSet.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}
```

```
KOŞUL BELİRTİLMEÐEN
ID: 1
NAME : Alice
AGE: 25
-----
ID: 2
NAME : Bob
AGE: 30
-----
ID: 3
NAME : Charlie
AGE: 22
-----
ID: 4
NAME : David
AGE: 28
-----
ID: 5
NAME : Eve
AGE: 35
-----
KOŞUL BELİRTİLDİKTEM SONRA
ID: 1
NAME : Alice
AGE: 25
-----
ID: 3
NAME : Charlie
AGE: 22
-----
Process finished with exit code 0
```

# JDBC - LIKE Clause Example

This chapter provides an example on how to select records from a table using JDBC application. This would add additional conditions using LIKE clause while selecting records from the table. Before executing the following example, make sure you have the following in place –

- To execute the following example you can replace the username and password with your actual user name and password.
- Your MySQL or whatever database you are using, is up and running.

## Required Steps

The following steps are required to create a new Database using 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.
- **Open a connection** – Requires using the `DriverManager.getConnection()` method to create a `Connection` object, which represents a physical connection with a database server.
- **Execute a query** – Requires using an object of type `Statement` for building and submitting an SQL statement to fetch records from a table which meet given condition. This Query makes use of **LIKE** clause to select records to select all the students whose first name starts with "za".
- **Clean up the environment** – try with resources automatically closes the resources.

```
import java.sql.*;

new *
public class LikeClauseExamples {
    new *
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/newbase23";
        String user = "root";
        String password = "1234";
        try (Connection connection = DriverManager.getConnection(url, user, password);
            Statement statement = connection.createStatement()) {
            ResultSet resultSet = statement.executeQuery("SELECT id, name, age FROM mytable05");
            System.out.println("LIKE KULLANILMADAN");
            while (resultSet.next()) {
                System.out.println("ID: " + resultSet.getInt(columnLabel: "id"));
                System.out.println("NAME : " + resultSet.getString(columnLabel: "name"));
                System.out.println("AGE: " + resultSet.getByte(columnLabel: "age"));
                System.out.println("-----");
            }

            resultSet = statement.executeQuery("SELECT * FROM mytable05 where name LIKE '%0%'");
            System.out.println("LIKE KULLANILDI");
            while (resultSet.next()) {
                System.out.println("ID: " + resultSet.getInt(columnLabel: "id"));
                System.out.println("NAME : " + resultSet.getString(columnLabel: "name"));
                System.out.println("AGE: " + resultSet.getByte(columnLabel: "age"));
                System.out.println("-----");
            }
            resultSet.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```
LIKE KULLANILMADAN
ID: 1
NAME : Alice
AGE: 25
-----
ID: 2
NAME : Bob
AGE: 30
-----
ID: 3
NAME : Charlie
AGE: 22
-----
ID: 4
NAME : David
AGE: 28
-----
ID: 5
NAME : Eve
AGE: 35
-----
LIKE KULLANILDI
ID: 2
NAME : Bob
AGE: 30
-----
Process finished with exit code 0
```

# JDBC - Sorting Data Example

This chapter provides an example on how to sort records from a table using JDBC application. This would use **asc** and **desc** keywords to sort records in ascending or descending order. Before executing the following example, make sure you have the following in place -

- To execute the following example you can replace the username and password with your actual user name and password.
- Your MySQL or whatever database you are using, is up and running.

## Required Steps

The following steps are required to create a new Database using 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.
- **Open a connection** – Requires using the DriverManager.getConnection() method to create a Connection object, which represents a physical connection with a database server.
- **Execute a query** – Requires using an object of type Statement for building and submitting an SQL statement to sort records from a table. These Queries make use of **asc** and **desc** clauses to sort data in ascending and descening orders.
- **Clean up the environment** – try with resources automatically closes the resources.

```
import java.sql.*;

new *
public class SortdeDataExamples01 {
    new *
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost/newbase23";
        String user = "root";
        String password = "1234";
        String query = "SELECT ID, NAME, AGE FROM MYTABLE05";
        try (Connection connection = DriverManager.getConnection(url, user, password);
            Statement statement = connection.createStatement()) {
            ResultSet resultSet = statement.executeQuery(query + " ORDER BY AGE ASC");
            System.out.println("ASC OLARAK SIRALANDI");
            while (resultSet.next()) {
                System.out.println("ID: " + resultSet.getInt(columnLabel.id));
                System.out.println("NAME : " + resultSet.getString(columnLabel.name));
                System.out.println("AGE: " + resultSet.getByte(columnLabel.age));
                System.out.println("-----");
            }
            System.out.println("DESC OLARAK SIRALANDI");
            resultSet = statement.executeQuery(query + " ORDER BY AGE DESC");
            while (resultSet.next()) {
                System.out.println("ID: " + resultSet.getInt(columnLabel.id));
                System.out.println("NAME : " + resultSet.getString(columnLabel.name));
                System.out.println("AGE: " + resultSet.getByte(columnLabel.age));
                System.out.println("-----");
            }
            resultSet.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

ASC OLARAK SIRALANDI

ID: 3  
NAME : Charlie  
AGE: 22

-----

ID: 1  
NAME : Alice  
AGE: 25

-----

ID: 4  
NAME : David  
AGE: 28

-----

ID: 2  
NAME : Bob  
AGE: 30

-----

ID: 5  
NAME : Eve  
AGE: 35

-----

DESC OLARAK SIRALANDI

ID: 5  
NAME : Eve  
AGE: 35

-----

ID: 2  
NAME : Bob  
AGE: 30

-----

ID: 4  
NAME : David  
AGE: 28

-----

ID: 1  
NAME : Alice  
AGE: 25

-----

ID: 3  
NAME : Charlie  
AGE: 22

-----

# JDBC - Update Records Example

This chapter provides an example on how to update records in a table using JDBC application. Before executing the following example, make sure you have the following in place -

- To execute the following example you can replace the username and password with your actual user name and password.
- Your MySQL or whatever database you are using is up and running.

## Required Steps

The following steps are required to create a new Database using 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.
- **Open a connection** – Requires using the `DriverManager.getConnection()` method to create a Connection object, which represents a physical connection with a database server.
- **Execute a query** – Requires using an object of type Statement for building and submitting an SQL statement to update records in a table. This Query makes use of **IN** and **WHERE** clause to update conditional records.
- **Clean up the environment** – `try with resources` automatically closes the resources.

### UPDATE ÖNCESİ

ID: 5

NAME : Eve

AGE: 35

-----

ID: 2

NAME : Bob

AGE: 30

-----

ID: 4

NAME : David

AGE: 28

-----

ID: 1

NAME : Alice

AGE: 25

-----

ID: 3

NAME : Charlie

AGE: 22

```
import java.sql.*;

new *
public class UpdateRecords01 {
    new *
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/newbase23";
        String user = "root";
        String password = "1234";
        String query = "SELECT id, name, age FROM MYTABLE05";

        try (Connection connection = DriverManager.getConnection(url, user, password);
             Statement statement = connection.createStatement()) {
            ResultSet resultSet = statement.executeQuery(query + " ORDER BY AGE DESC");
            System.out.println("UPDATE ÖNCESİ");
            while (resultSet.next()) {
                System.out.println("ID: " + resultSet.getInt(columnLabel: "id"));
                System.out.println("NAME : " + resultSet.getString(columnLabel: "name"));
                System.out.println("AGE: " + resultSet.getByte(columnLabel: "age"));
                System.out.println("-----");
            }
            statement.executeUpdate(query: "UPDATE MYTABLE05 SET AGE = 100 WHERE id IN (2,3)");
            resultSet = statement.executeQuery(query + " ORDER BY AGE DESC");
            System.out.println("UPDATE SONRASI");
            while (resultSet.next()) {
                System.out.println("ID: " + resultSet.getInt(columnLabel: "id"));
                System.out.println("NAME : " + resultSet.getString(columnLabel: "name"));
                System.out.println("AGE: " + resultSet.getByte(columnLabel: "age"));
                System.out.println("-----");
            }
            resultSet.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

### UPDATE SONRASI

ID: 2

NAME : Bob

AGE: 100

-----

ID: 3

NAME : Charlie

AGE: 100

-----

ID: 5

NAME : Eve

AGE: 35

-----

ID: 4

NAME : David

AGE: 28

-----

ID: 1

NAME : Alice

AGE: 25