

Type I Driver — JDBC-ODBC Bridge Driver

1. It Converts JDBC calls -> ODBC calls -> database.
2. Requires ODBC driver installed on the client.
3. It is simple to use.
4. It is Very slow (two-level translation).
5. It is Not portable (platform-dependent).
6. It is Removed from Java 8+.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
Connection con = DriverManager.getConnection("jdbc:odbc:mydsn");
```

Type II Driver — Native-API → Java Driver

1. Converts JDBC calls to native database API (C/C++ libraries).
2. We need to install native client libraries on the machine.
3. It is Faster than Type I.
4. It Requires DB vendor native libraries.
5. Not platform independent.

(Oracle OCI driver)

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
Connection con = DriverManager.getConnection(  
    "jdbc:oracle:oci:@mydb", "user", "pass");
```

Type III Driver — These are Network Protocol Driver (Middleware Driver)

1. In this driver JDBC calls → middleware server → database.
 2. Middleware translates JDBC into DB-specific protocol.
 3. No client-side DB libraries required.
 4. It can access multiple databases using a single driver.
 5. Slightly slower than Type IV because of extra network layer.
- JDBC calls go to middleware like:
 - WebLogic
 - WebSphere
 - DBAnywhere

```
Connection con = DriverManager.getConnection(
    "jdbc:weblogic:mydb", "user", "pass");
```

Type IV Driver — Thin Driver (Pure Java Driver)

1. JDBC API → directly communicates with database-specific protocol
2. It is pure Java specific.
3. It is Most commonly used
4. It is Fastest driver.
5. Database independent, no native libraries are needed.
6. Always used in web apps and webservice
7. We use One driver per database (Oracle, MySQL, PostgreSQL etc.)

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

```
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/test", "root", "password");
```

JDBC ---Java database Connection

Step1 :

Register Driver

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

```
DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver())
```

Step2:

Get connection--- to get the connection to the database

```
Connection conn=DriverManager.getConnection(url,username,password)
```

url=→ jdbc:mysql://hostname:port/databasename

Step3:

Create statement

3 types of statement

1. Statement---→

```
Statement st=conn.createStatement()
```

 - a. This statement can be used to execute different queries
 - b. It is vulnerable for SQL injection attack

2. PreparedStatement -> PreparedStatement pst=conn.prepareStatement(query)
 - a. This statement can be used to execute only query associated with it
 - b. It is more secure than createStatement
 - c. Before execution of the query, It is necessary to set values for '?' which is a placeholder
3. CallableStatement-->


```
CallableStatement cst=conn.preopareCall({call <Procedurename>(arguments)})
```

 - a. Arguments can be used as '?'
 - b. At the time execution set the value for '?' placeholder
 - c. For in type parameter we use setter method
 - d. And for out type parameters we use registerOutParameter
 - e. To retrieve the value of out type parameter we use getter method

Step 4:

Execute the statement

1. To execute query with select statement
 - a. Then use executeQuery() , it returns ResultSet
 - b. Then we use while loop or if statement with rs.next(), to navigate through ResultSet
 - c. rs.next() returns true, if next row exists, else it returns false
 - d. and to retrieve the fields of current record, use getXXX methods, example: getInt, getString
2. To execute DML operation, like insert, update, delete, then use executeUpdate(), This function returns int value, i.e number of rows affected
3. To execute procedure, use execute() function, it returns true /false

Step 5:

Close connection-> once the database usage is done , then close the connection

Conn.close();

Singleton pattern

The Singleton Pattern is a creational design pattern in software engineering that ensures:

Only one instance of a class is created in the entire application.

That single instance is globally accessible.

```
public class Singleton {
    private static final Singleton instance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
```

```
        return instance;
    }
}
```

Factory Pattern

If we write a function in a class, which creates Objects of some other class, then the method uses factory pattern

`ExecutorService es=Executors.getFixedThreadPool()`-→this function uses factory pattern

Factory Pattern provides an interface for creating objects, but allows subclasses or separate factory classes to decide **which class to instantiate**.

It helps you:

- Remove tight coupling between classes.
- Centralize object creation logic.
- Make your code more flexible and maintainable.

```
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

class Rectangle implements Shape {
    public void draw() {
        System.out.println("Drawing Rectangle");
    }
}

class Square implements Shape {
    public void draw() {
        System.out.println("Drawing Square");
    }
}

class ShapeFactory {

    public Shape getShape(String type) {
        if (type == null) return null;

        switch (type.toLowerCase()) {
```

```
        case "circle": return new Circle();
        case "rectangle": return new Rectangle();
        case "square": return new Square();
        default: return null;
    }
}
}
```

```
public class FactoryDemo {
    public static void main(String[] args) {
        ShapeFactory factory = new ShapeFactory();

        Shape s1 = factory.getShape("circle");
        s1.draw();

        Shape s2 = factory.getShape("square");
        s2.draw();
    }
}
```