Training course

JF3-5 Java Persistence with JDBC

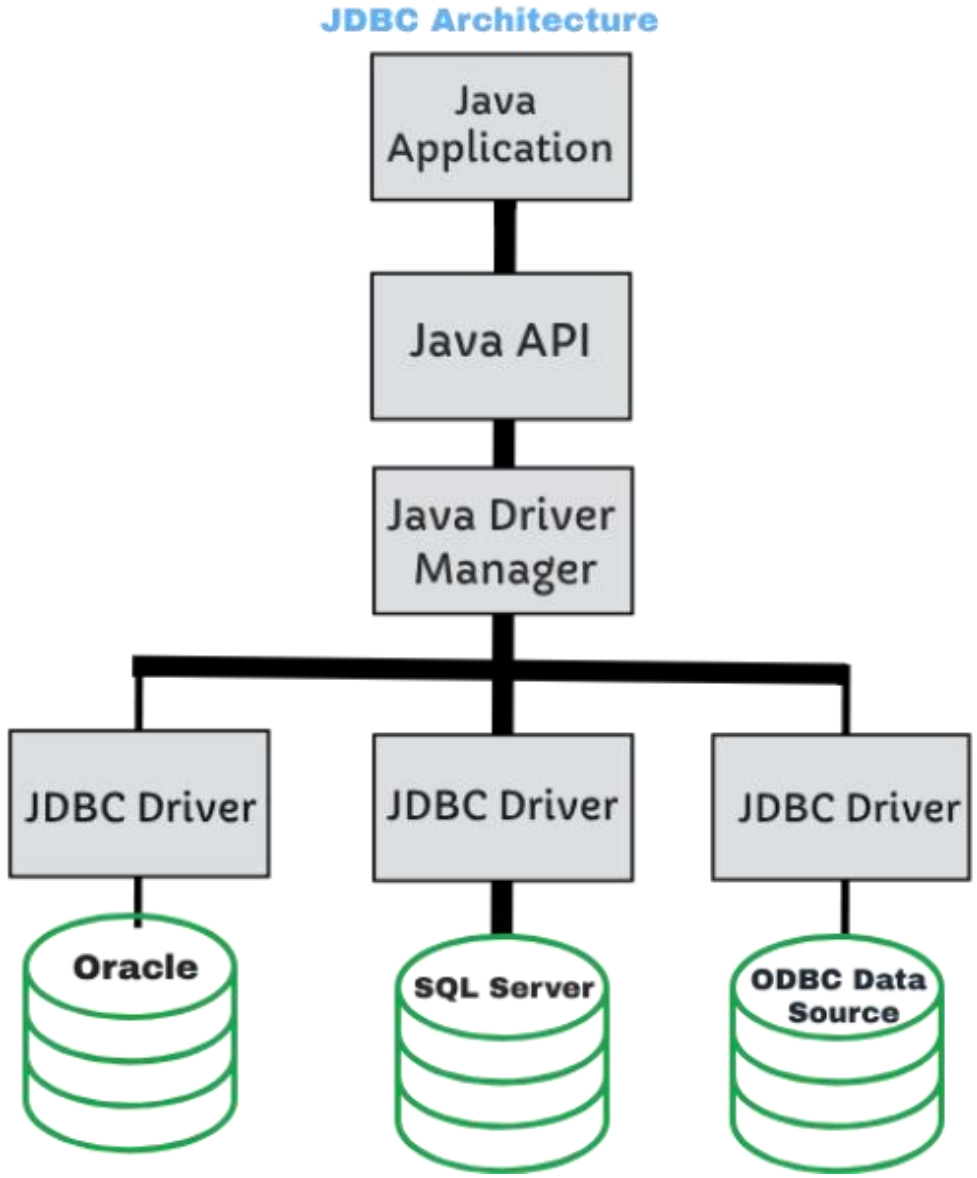
Data persistence

Data Persistence is a means for an application to persist and retrieve information from a non-volatile storage system. Persistence is vital to enterprise applications because of the required access to relational databases. Applications that are developed for this environment must manage persistence themselves or use third-party solutions to handle database updates and retrievals with persistence.

- JDBC = Java Database Connectivity
- API for connecting and executing queries on a database
- JDBC can work with any database as long as proper drivers are provided

JDBC Drivers

A JDBC driver is a JDBC API implementation used for connecting to a particular type of database. There are several types of JDBC drivers.



Type 1 : JDBC-ODBC bridge driver

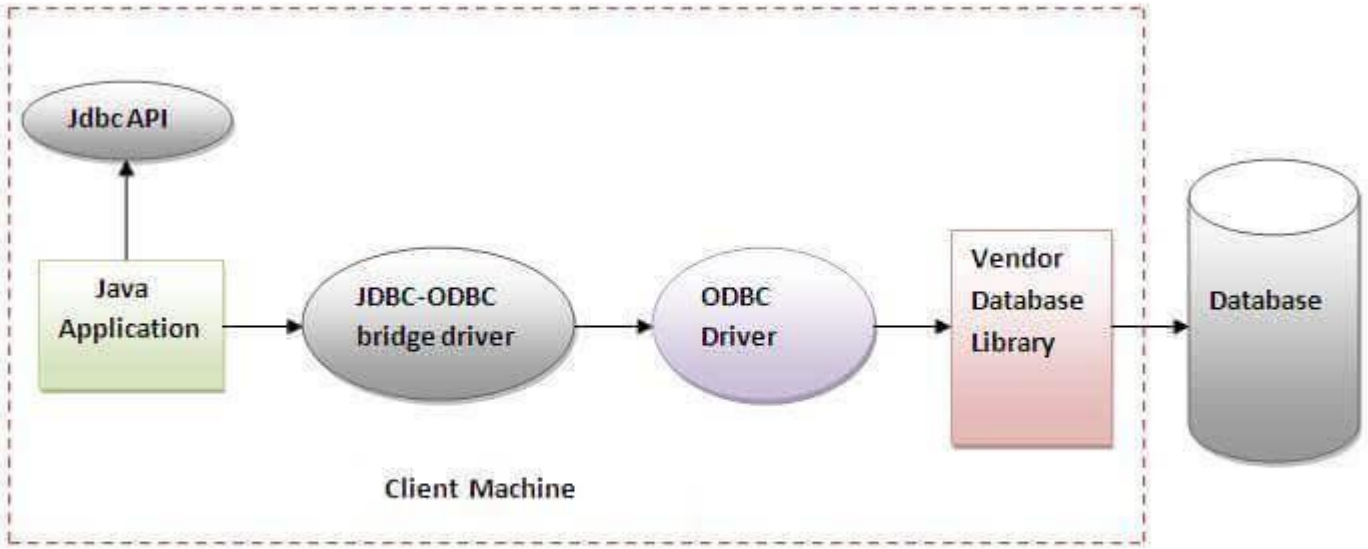


Figure- JDBC-ODBC Bridge Driver

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

Type 2 : Native-API driver

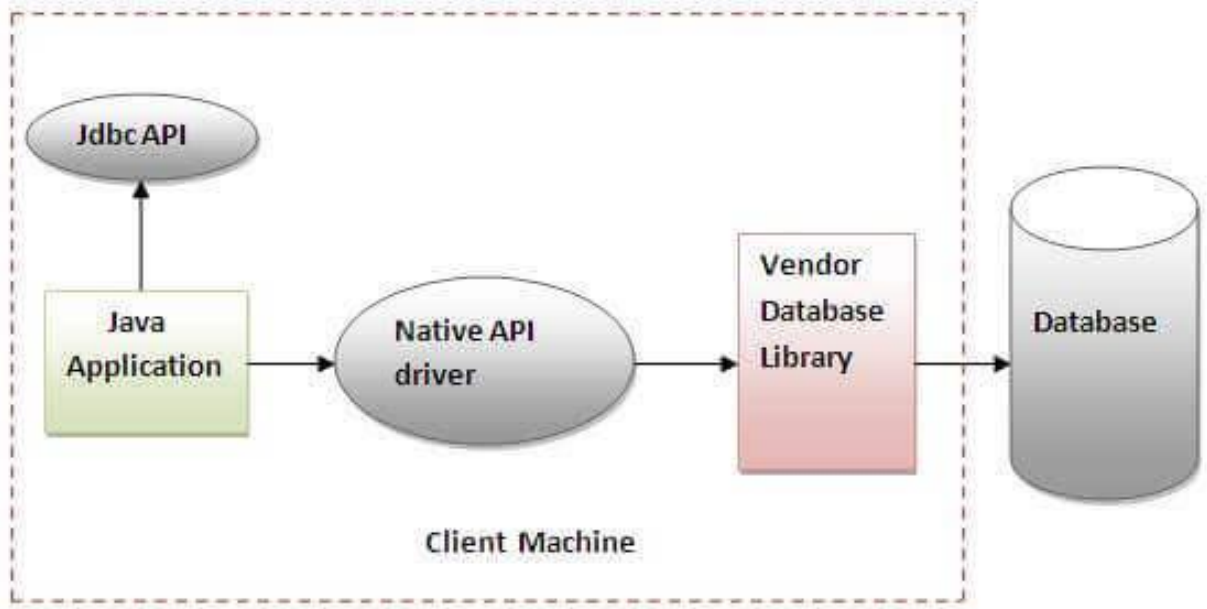


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

Type 3 : Network Protocol driver

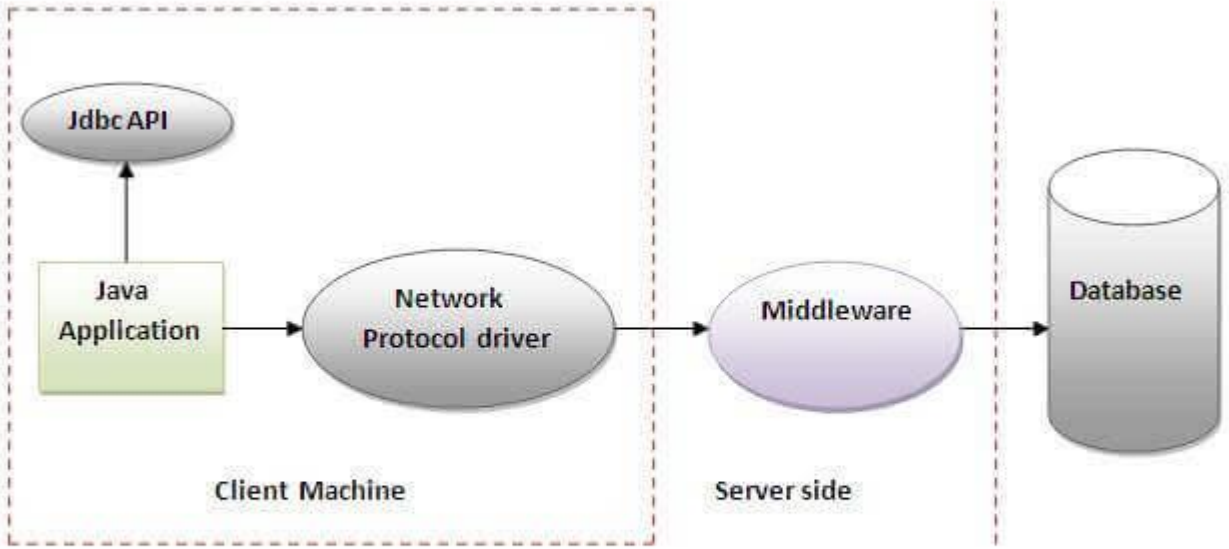


Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Type 4 : Thin driver

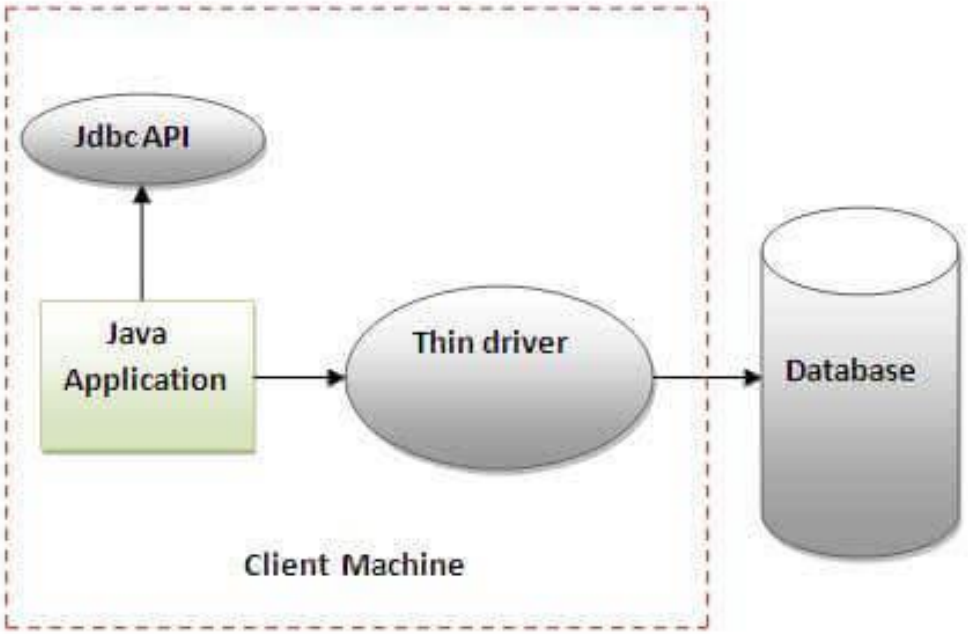


Figure- Thin Driver

Advantage:

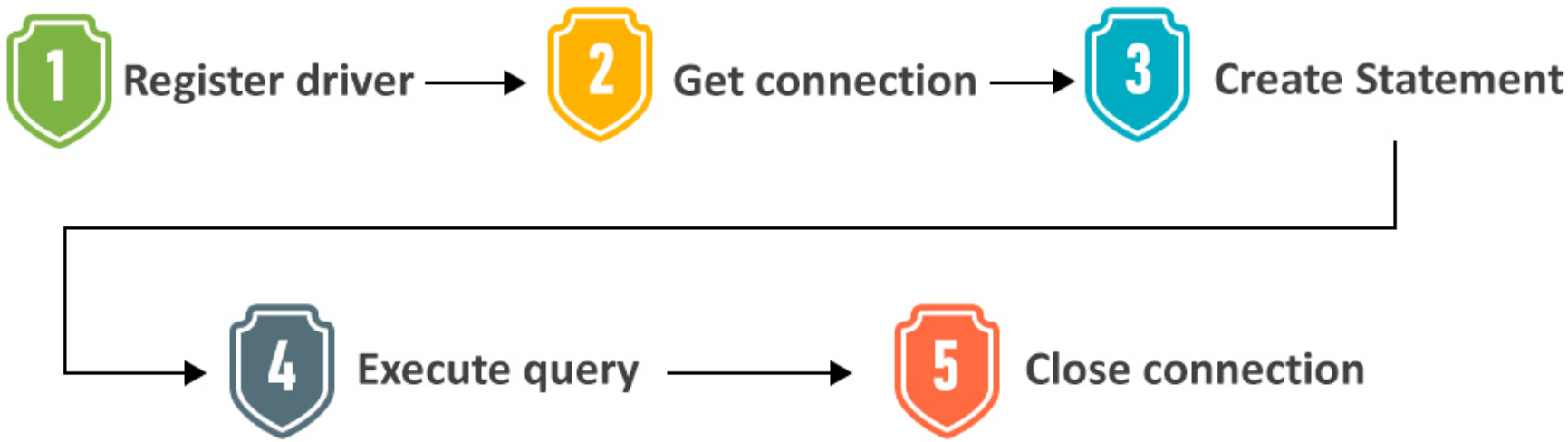
- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

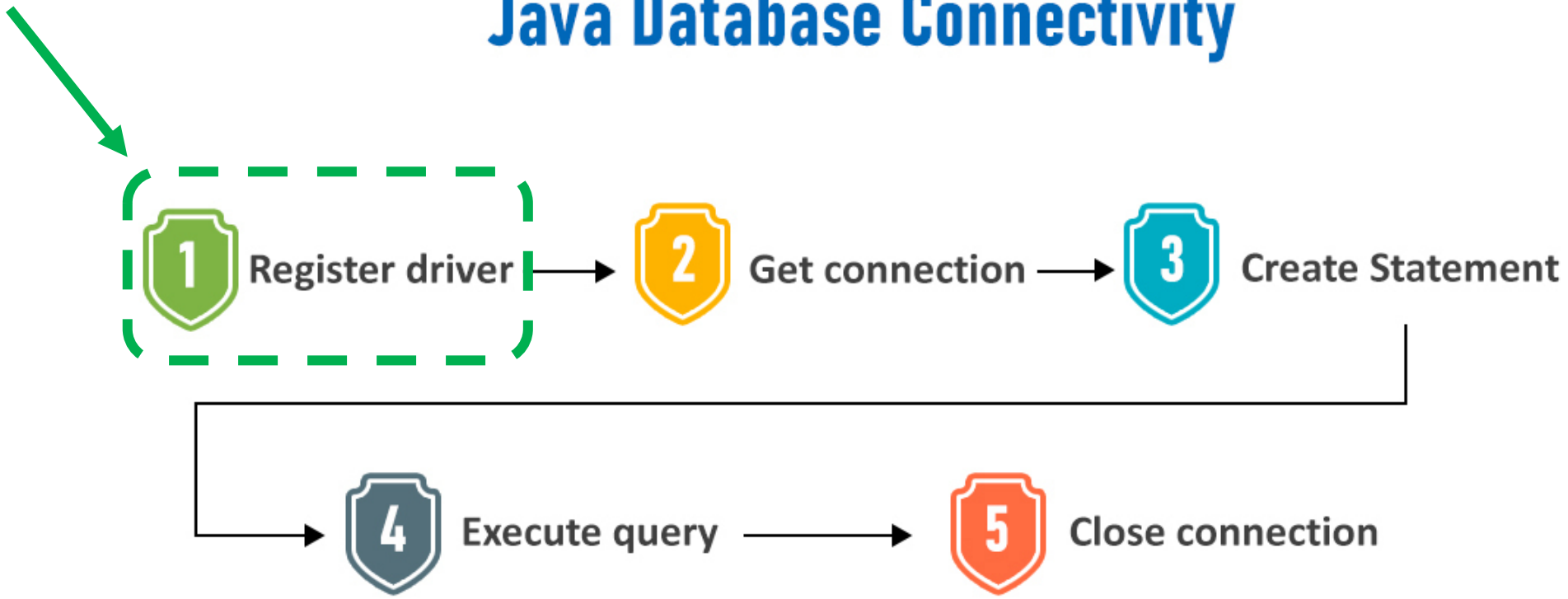
- Drivers depend on the Database.

The most commonly used type is **type 4**, as it has the advantage of being **platform-independent**. Connecting directly to a database server provides better performance compared to other types. The downside of this type of driver is that it's database-specific – given each database has its own specific protocol.

Java Database Connectivity



Java Database Connectivity

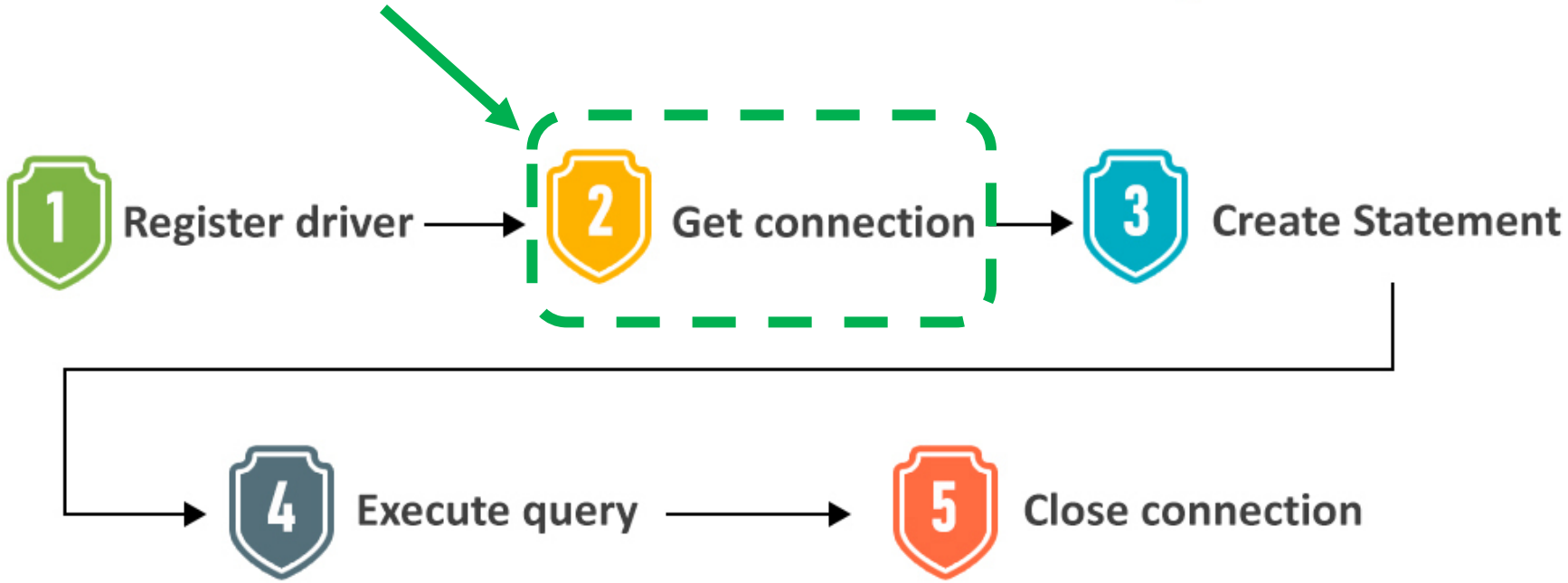


Registering the Driver (Type 4)

- Since we're using a Postgresql database, we need the Postgres connector dependency:

```
<dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.5.1</version>
</dependency>
```

Java Database Connectivity



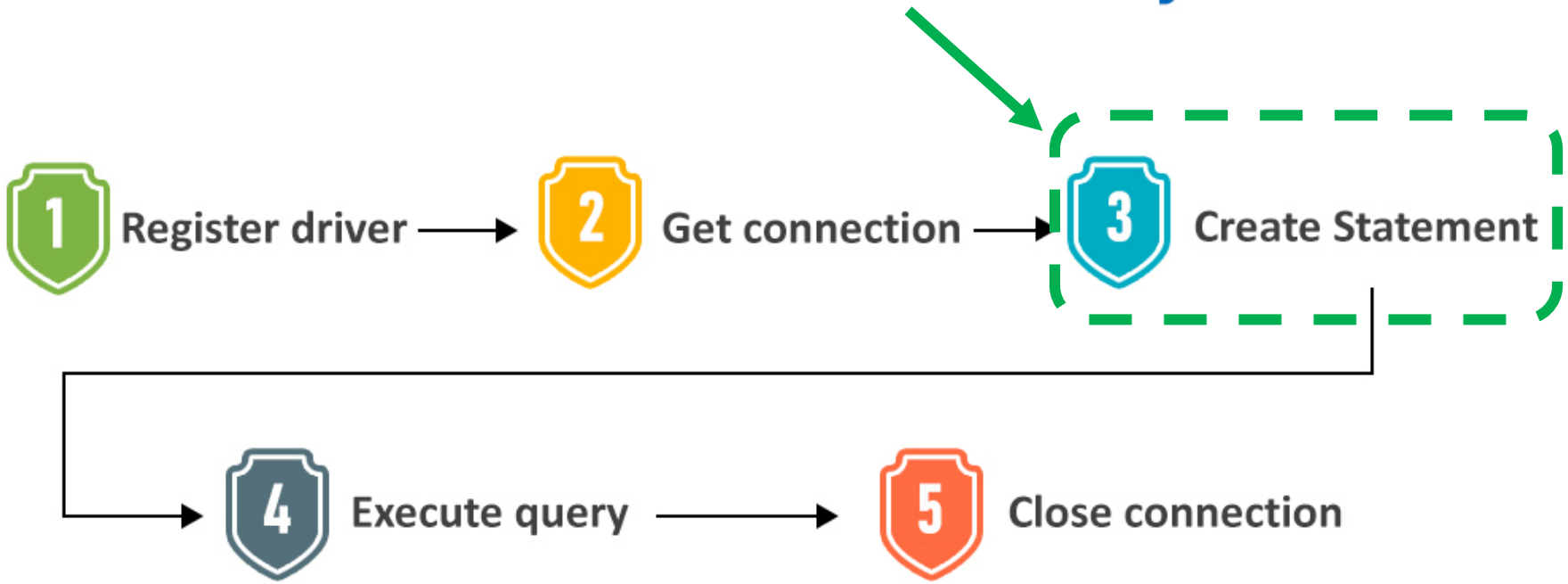
Creating the Connection

- To open a connection, we can use the *getConnection()* method of *DriverManager* class. This method requires a connection URL *String* parameter

```
try (Connection connection = DriverManager
  .getConnection("jdbc:postgresql://localhost:5432/demo", "user1", "pass")) {
    // use connection here
}
```

- Since the Connection is an **AutoCloseable** resource, we should use it inside a **try-with-resources block**

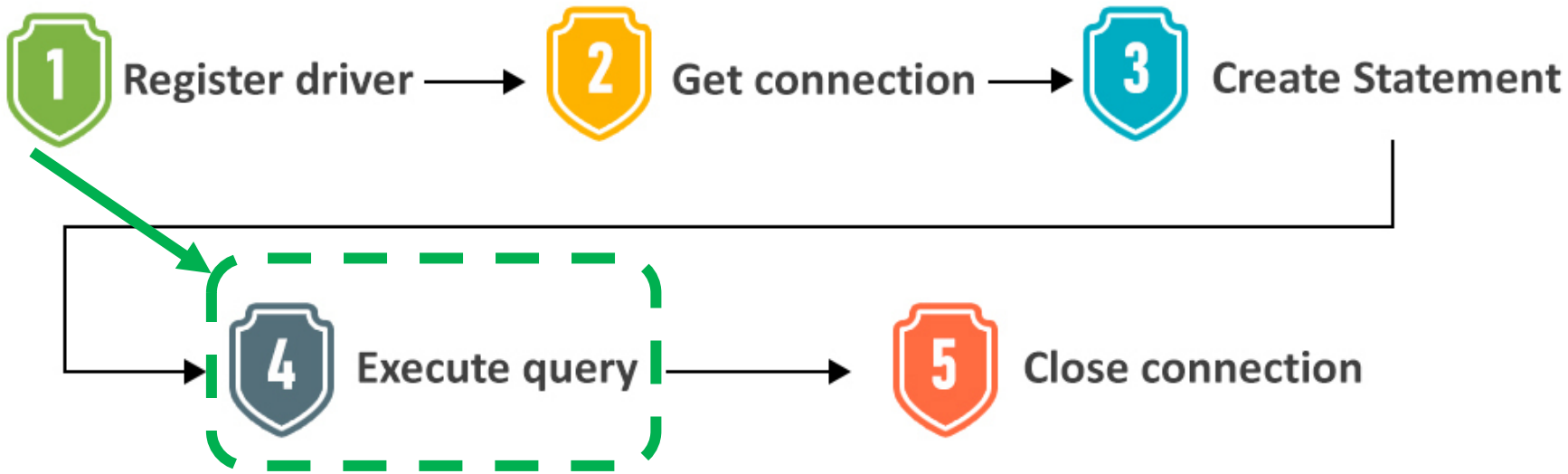
Java Database Connectivity



Create Statements

To send SQL instructions to the database, we can use instances of type **Statement**, **PreparedStatement**, or **CallableStatement**, which we can obtain using the Connection object.

Java Database Connectivity



Statement

```
try (Statement stmt = connection.createStatement()) {
    String tableSql = “CREATE TABLE users (id int PRIMARY, name varchar(30))”;
    stmt.execute(tableSql);
}
```

Executing SQL instructions can be done through the use of three methods:

- **executeQuery()** for SELECT instructions
- **executeUpdate()** for updating the data or the database structure
- **execute()** can be used for both cases above when the result is unknown

Statement

- When using the execute() method to update the data, then the stmt.getUpdateCount() method returns the number of rows affected.

- If the result is 0 then either no rows were affected, or it was a database structure update command.

- If the value is -1, then the command was a SELECT query; we can then obtain the result using stmt.getResultSet().

Statement - Read SELECT query

```
String selectSql = "SELECT * FROM users";
try (ResultSet resultSet = stmt.executeQuery(selectSql)) {
    List<User> users = new ArrayList<>();
    while (resultSet.next()) {
        User user = new User();
        user.setId(resultSet.getInt("id"));
        user.setName(resultSet.getString("name"));
        users.add(user);
    }}
```

Statement - ResultSet

To navigate this type of ResultSet, we can use one of the methods:

- first(), last(), beforeFirst(), beforeLast() – to move to the first or last line of a ResultSet or to the line before these
- next(), previous() – to navigate forward and backward in the ResultSet
- getRow() – to obtain the current row number
- moveToInsertRow(), moveToCurrentRow() – to move to a new empty row to insert and back to the current one if on a new row
- absolute(int row) – to move to the specified row
- relative(int nrRows) – to move the cursor the given number of rows

Statement - ResultSet

To persist the ResultSet changes to the database, we must further use one of the methods:

- updateRow() – to persist the changes to the current row to the database
- insertRow(), deleteRow() – to add a new row or delete the current one from the database
- refreshRow() – to refresh the ResultSet with any changes in the database
- cancelRowUpdates() – to cancel changes made to the current row

PreparedStatement

PreparedStatement objects contain precompiled SQL sequences. They can have one or more parameters denoted by a question mark.

```
String updatePositionSql = "UPDATE users SET name=? WHERE id=?";
try (PreparedStatement pstmt = con.prepareStatement(updateNameSql)) {
    pstmt.setString(1, “Paul");
    pstmt.setInt(2, 1);
    pstmt.executeUpdate();
}
```

CallableStatement

The CallableStatement interface allows calling stored procedures.

```
String preparedSql = "{call insertEmployee(?,?,?,?)}";
try (CallableStatement cstmt = con.prepareCall(preparedSql)) {
    cstmt.setString(2, "ana");
    cstmt.setString(3, "tester");
    cstmt.setDouble(4, 2000);
    cstmt.execute();
}
```

```
delimiter //
CREATE PROCEDURE insertEmployee(OUT emp_id int,
  IN emp_name varchar(30), IN position varchar(30), IN salary double)
BEGIN
INSERT INTO employees(name, position,salary) VALUES (emp_name,position,salary);
SET emp_id = LAST_INSERT_ID();
END //
delimiter ;
```

Handling Transactions

- By default, each SQL statement is committed right after it is completed. However, it's also possible to **control transactions programmatically**.

- This may be necessary in cases when we want to preserve data consistency, for example when we only want to commit a transaction if a previous one has completed successfully.

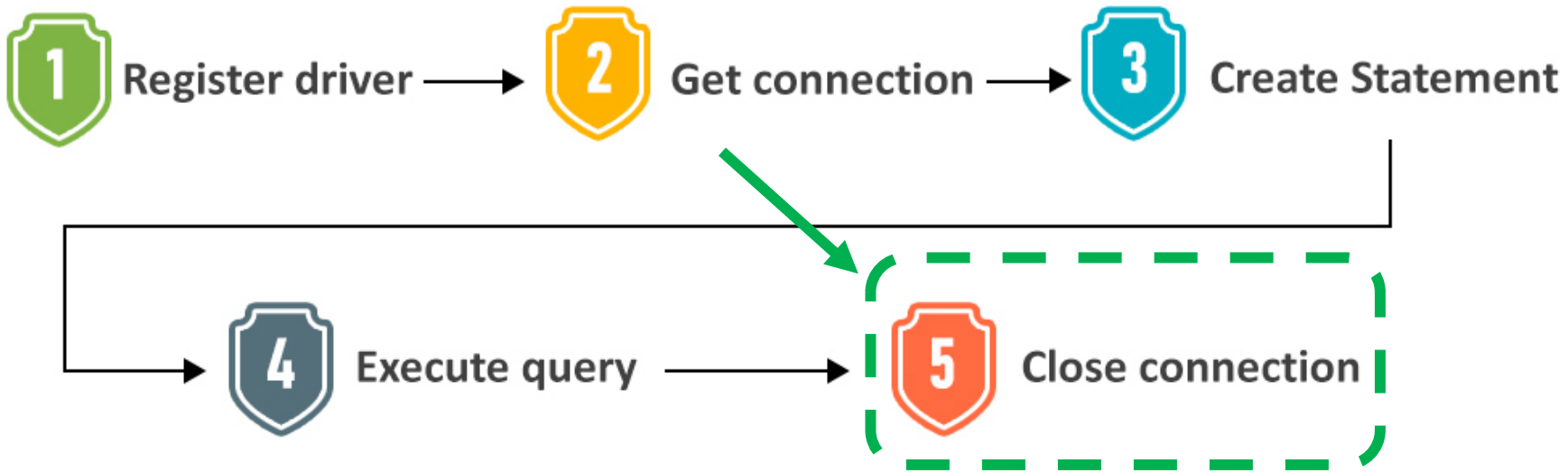
- First, we need to set the ***autoCommit*** property of ***Connection*** to *false*, then use the ***commit()*** and ***rollback()*** methods to control the transaction.

```
String updatePositionSql = "UPDATE employees SET position=? WHERE emp_id=?";
PreparedStatement pstmt = con.prepareStatement(updatePositionSql);
pstmt.setString(1, "lead developer");
pstmt.setInt(2, 1);

String updateSalarySql = "UPDATE employees SET salary=? WHERE emp_id=?";
PreparedStatement pstmt2 = con.prepareStatement(updateSalarySql);
pstmt2.setDouble(1, 3000);
pstmt2.setInt(2, 1);

boolean autoCommit = con.getAutoCommit();
try {
    con.setAutoCommit(false);
    pstmt.executeUpdate();
    pstmt2.executeUpdate();
    con.commit();
} catch (SQLException exc) {
    con.rollback();
} finally {
    con.setAutoCommit(autoCommit);
}
```

Java Database Connectivity



Closing the Resources

- When we're no longer using it, we need to close the connection to release database resources.

- However, if we're using the resource in a try-with-resources block, we don't need to call the **close()** method explicitly, as the try-with-resources block does that for us automatically.

- If you don't close it, it leaks, and ties up server resources

Assignment