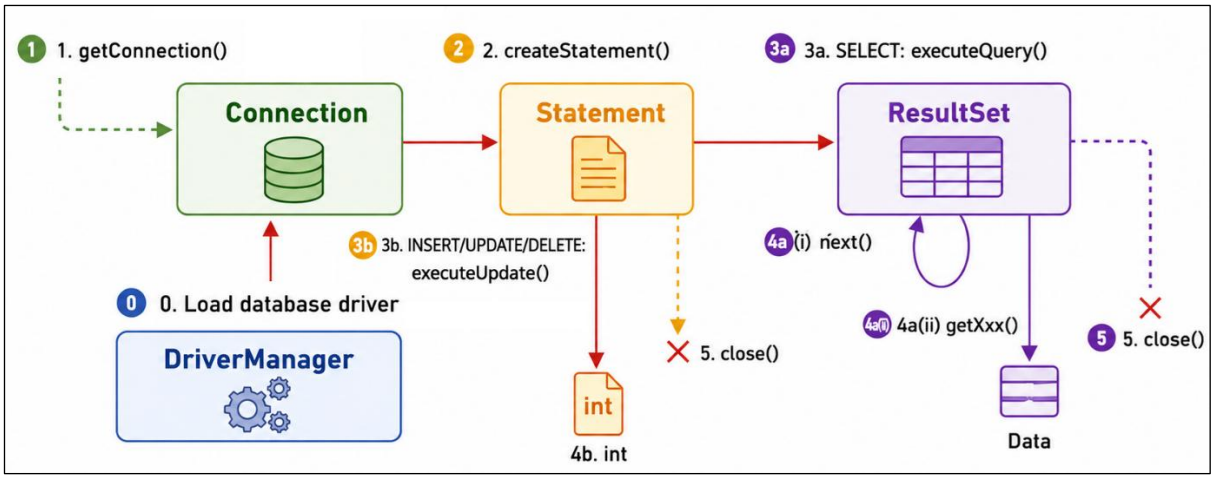
MySQL Connector/J (`mysql-connector-java-x.x.xx.jar`) herunterladen und einbinden

1. Die JAR-Datei per Drag & Drop in den Projektordner ziehen und im Dialog **Copy files** bestätigen.
2. Rechtsklick auf die JAR-Datei → **Build Path** → **Configure Build Path…**
3. Im Tab **Libraries** den **Modulepath** auswählen und auf **Add JARs…** klicken.
4. Den MySQL Connector aus dem Projektverzeichnis auswählen und mit **OK** bestätigen.
5. Änderungen mit **Apply and Close** speichern. Fertig!




```
①  import java.sql.*;
②  public class mySQL_Basic {
③  public static void main(String[] args) throws SQLException {
④     Connection con;
⑤     Statement stm;
⑥     ResultSet res;
⑦     // 0. Load database driver und 1. getConnection()
      con = DriverManager.getConnection("jdbc:mysql://host/base","user","pass");
⑧     // 2. createStatement
      stm = con.createStatement();
⑨     // 3a. executeQuery (SELECT)
      res = stm.executeQuery("SELECT att1, att2, ... FROM tab");
⑩     // 4a Über Ergebnis itterieren (next -> getXXX)
      while (res.next()) {
⑪        String att1 = res.getString("att1");
⑫        int att2 = res.getInt("att2"); // ggf. weitere Attribute
⑬     }
⑭     con.close();   // 5. Verbindung trennen
⑮  }  }
```

Die meisten Methoden können eine **SQLException** werfen und müssen mit **throws …** oder **try{…} catch{…}** abgefangen werden!

Auslesen und Sichern des **ResultSet** mit **next()** und den passenden **getXXX()** Methoden.

Simples Beispiel zur Datenbankanbindung

Alle Datenänderungen werden mit `executeUpdate()` ausgeführt (SELECT gehört nicht dazu).

```
⑨  // 3b. executeUpdate (INSERT | UPDATE | DELETE)
:  String SQL = "INSERT INTO ... | UPDATE ... | DELETE FROM ...)";
⑬  int Anzahl = stm.executeUpdate(SQL); // 4b. liefert Anzahl der betroffenen Datensätze
```

SQL als Prepared Statement ausführen

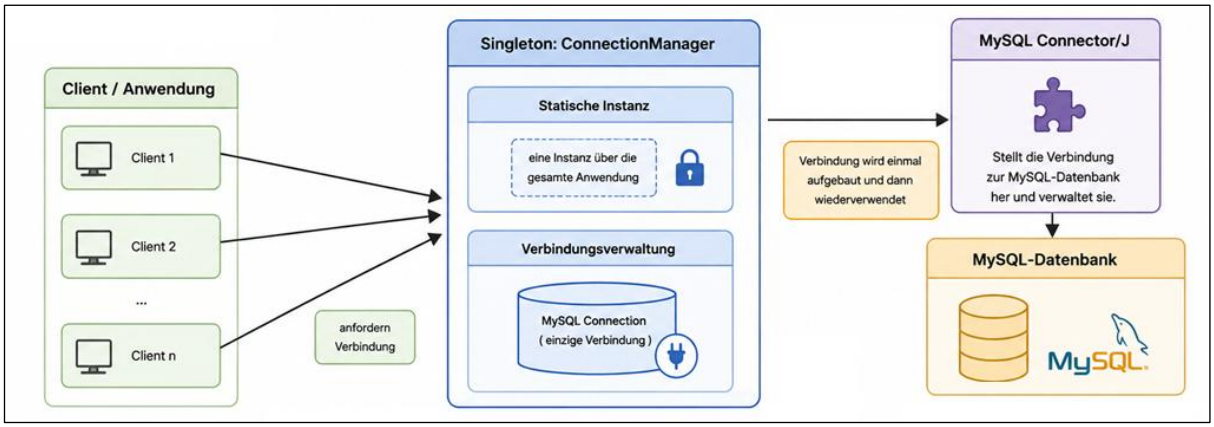
Ein Prepared Statement ist eine vorbereitete SQL-Abfrage. Werte werden erst später gebunden, wodurch Zugriffe sicherer (Schutz vor SQL-Injection) und effizienter werden. Es eignet sich besonders für Abfragen mit variablen Eingaben.

```
⑧  // SQL-DML oder DQL mit Platzhalter (?) vorbereiten
   SQL = "INSERT INTO tab (att1, att2) VALUES( ? , ? )";
:  PreparedStatement preStm = Connection.prepareStatement(SQL); // preparedState erzeugen
:  preStm.setXXX(position, value);                  // allen Platzhalter Parameter zuweisen
⑬  int Anzahl = preStm.executeUpdate();                                  // SQL ausführen
```

Verbindung mit Singleton - Pattern

Ein Singleton-Connection-Manager in Java ist bei einer MySQL-Datenbank sinnvoll, weil dadurch nur **eine zentrale Instanz** für die Verwaltung der Datenbankverbindungen existiert. Das bietet mehrere Vorteile:

- **Ressourcenschonung:** Es werden nicht unnötig viele Verbindungen zur Datenbank geöffnet.
- **Zentrale Verwaltung:** Zugangsdaten und Verbindungslogik müssen nur an einer Stelle gepflegt werden.
- **Konsistenz:** Alle Teile der Anwendung nutzen dieselbe Verwaltungsinstanz.




```
①  public class ConnectionManager {
②     private static Connection con = null;
③     public static Connection getInstance() throws SQLException {
④        if (con == null) {
⑤           con = DriverManager.getConnection("jdbc:mysql://host/base","user","pass");
⑥        }
⑦        return con;
⑧     }
⑨     public static void closeInstance() throws SQLException  {
⑩        if (con != null) {
⑪           con.close(); con = null;
⑫  }  }  }
```

Auch hier können **SQLException** geworfen werden und müssen mit **throws …** (oder besser) mit **try{…} catch{…}** abgefangen werden!

Singleton Pattern zum Verbindungsauf- und Abbau

In der Ausführungsklasse das Verbindungsobjekt erstellen mit:

`Connection con = MySQL_SingletonPattern.getInstance();`

bzw. Verbindung trennen mit: **`MySQL_SingletonPattern.closeInstance()`**