JAVA DEVELOPER INTERNSHIP BY ELEVATE LABS

Interview Questions:

1.What is JDBC?
2.What is PreparedStatement?
3.Difference between Statement and PreparedStatement?
4.How do you handle SQL exceptions?
5.How to prevent SQL Injection?
6.What is JDBC DriverManager?
7.How to close connections?
8.What is a ResultSet?
9.What is auto-commit in JDBC?
10.How to connect Java to MySQL

1) What is JDBC?

Explanation (in depth)

JDBC (Java Database Connectivity) is Java's standard API for talking to relational databases. It defines a set of interfaces (e.g., Connection, Statement, PreparedStatement, ResultSet, Driver) that drivers implement for each database (MySQL, PostgreSQL, etc.).

Key ideas:

- **Abstraction layer**: Your Java code calls JDBC interfaces; a vendor-specific **JDBC driver** translates those calls to the database's wire protocol.
- **Core workflow**: Load/register driver → obtain a Connection → create statements → execute SQL → read ResultSet → commit/rollback → close.
- **Portability**: Swap drivers/URLs and most code still works.
- **Features**: Transactions, batching, streaming results, metadata (DatabaseMetaData, ResultSetMetaData).

Sample code

```
try (Connection conn = DriverManager.getConnection(
      "jdbc:mysql://localhost:3306/employee_db", "root", "password");
    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery("SELECT id, name FROM employees")) {
   while (rs.next()) {
     System.out.println(rs.getInt("id") + " - " + rs.getString("name"));
   }
}
```

Real-life examples

- Building an **Employee Management** tool that adds/updates staff records in MySQL.
- A **Billing** service fetching invoices from PostgreSQL while reusing the same JDBC code structure.

2) What is PreparedStatement?

Explanation (in depth)

PreparedStatement is a precompiled, parameterized SQL statement. You write SQL with ? placeholders and bind typed values via setters (setInt, setString, etc.).

Why it matters:

- **Security**: Prevents SQL injection by separating SQL from data.
- **Performance**: DB can reuse the compiled query plan across executions.
- **Type safety**: JDBC driver handles quoting/escaping and type conversion.
- **Batching**: Efficiently execute many similar operations with addBatch()/executeBatch().

Sample code

```
String sql = "INSERT INTO employees(name, position, salary) VALUES(?, ?, ?)";
try (PreparedStatement ps = conn.prepareStatement(sql)) {
   ps.setString(1, "John");
   ps.setString(2, "Manager");
   ps.setDouble(3, 55000.0);
   ps.executeUpdate();
}
```

Real-life examples

- Importing a **CSV of 10,000 employees** with one prepared SQL and batching for speed.
- A **login** form querying SELECT * FROM users WHERE email = ? safely.

3) Difference between Statement and PreparedStatement?

Explanation (in depth)

- **Statement**: Executes static SQL text. You concatenate values into the SQL string. Fast to write, risky (SQL injection), no plan reuse.
- **PreparedStatement**: SQL with ? parameters; values bound separately. Safer, usually faster for repeated execution, supports batching and typed binding.

Other notes:

- Prepared statements often enable **server-side** prepares & plan caching.
- PreparedStatement simplifies handling quotes/locale/escaping for strings, dates, etc.
- Use Statement for one-off admin/DDL (e.g., CREATE TABLE) or when no parameters.

Sample code (contrast)

```
// Risky (Statement)
String name = "Alice"; // if user input: dangerous
String q1 = "SELECT * FROM employees WHERE name = '" + name + "'";
try (Statement st = conn.createStatement(); ResultSet rs = st.executeQuery(q1)) {
/*...*/ }

// Safe (PreparedStatement)
String q2 = "SELECT * FROM employees WHERE name = ?";
try (PreparedStatement ps = conn.prepareStatement(q2)) {
   ps.setString(1, name);
   try (ResultSet rs = ps.executeQuery()) { /*...*/ }
}
```

Real-life examples

- **Search box** on “Employees by title” → PreparedStatement avoids breaking on names like O'Neil.
- **Schema migration** script using Statement to run ALTER TABLE commands without parameters.

4) How do you handle SQL exceptions?

Explanation (in depth)

All JDBC operations can throw SQLException. Handle them by:

- **Logging diagnostics**: getSQLState(), getErrorCode(), getMessage(), getCause().
- **Categorizing**: timeouts (SQLTimeoutException), integrity violations (SQLIntegrityConstraintViolationException), transient vs. non-transient.
- **Transactional safety**: on failure inside a manual transaction, rollback().
- **Propagation**: wrap in a domain exception or rethrow after cleanup.
- **Retry**: optionally for transient errors (e.g., deadlocks), with backoff.

Sample code

```
try {
   conn.setAutoCommit(false);
   // ... SQL updates ...
   conn.commit();
} catch (SQLIntegrityConstraintViolationException e) {
   conn.rollback();
   System.err.println("Duplicate key or FK violation: " + e.getMessage());
} catch (SQLTimeoutException e) {
   conn.rollback();
   System.err.println("Database timed out: " + e.getMessage());
} catch (SQLException e) {
   conn.rollback();
   System.err.printf("SQLState=%s Code=%d Msg=%s%n",
         e.getSQLState(), e.getErrorCode(), e.getMessage());
} finally {
   conn.setAutoCommit(true);
}
```

Real-life examples

- Preventing **partial payroll** updates: if any employee update fails, rollback the whole batch.
- Detecting **duplicate email** on registration and returning a friendly “Email already used”.

5) How to prevent SQL Injection?

Explanation (in depth)

- **Always use PreparedStatement** with ? placeholders.
- **Allowlist inputs** when filtering on enums (e.g., role, sort column).
- **Least-privilege** DB user; avoid powerful accounts.
- **Avoid string concatenation** for SQL.
- **Optional**: Stored procedures with parameters (still validate inputs).
- **Input validation**: check format (emails, numbers) before hitting DB.

Sample code

```
String sql = "SELECT id, name FROM employees WHERE position = ? AND
salary >= ?";
try (PreparedStatement ps = conn.prepareStatement(sql)) {
   ps.setString(1, userProvidedPosition);
   ps.setDouble(2, minSalary);
   try (ResultSet rs = ps.executeQuery()) { /*...*/ }
}
```

Real-life examples

- A **search API** that takes department and minSalary—all bound via PreparedStatement.
- An **admin grid** that only allows sorting by a predefined set of column names (allowlist) to prevent injected ORDER BY clauses.

6) What is JDBC DriverManager?

Explanation (in depth)

DriverManager is the legacy JDBC service that discovers registered drivers (via the Service Provider mechanism) and hands out Connection objects from getConnection(url, user, pass).

Notes:

- With modern drivers, you **don't need** to manually Class.forName—the driver auto-registers.
- In production, prefer a **DataSource with a connection pool** (e.g., HikariCP) for performance and control; but DriverManager is fine for simple apps.

Sample code

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

Real-life examples

- A **CLI utility** that runs one report monthly—simple DriverManager is enough.
- A **small desktop app** (no app server) connecting directly to MySQL without pooling.

7) How to close connections?

Explanation (in depth)

- Use **try-with-resources** so JDBC objects close automatically in the correct order (ResultSet → Statement → Connection).
- If not using try-with-resources, close in finally.
- Closing a Connection returns it to the pool (if pooled) or closes the socket.
- Don't reuse closed objects; always create fresh statements.

Sample code (best practice)

```
String sql = "SELECT * FROM employees";

try (Connection conn = DriverManager.getConnection(URL, USER, PASS);

    PreparedStatement ps = conn.prepareStatement(sql);

    ResultSet rs = ps.executeQuery()) {

   while (rs.next()) { /*...*/ }

}

// all closed automatically here
```

Real-life examples

- A **web endpoint** that fetches employees—each request uses try-with-resources to avoid leaks.
- A **batch job** iterating thousands of rows—processing inside the try-with-resources block prevents "too many connections" errors.

8) What is a ResultSet?

Explanation (in depth)

ResultSet is a cursor over rows returned by a query.
Capabilities & options:

- **Navigation type**: TYPE_FORWARD_ONLY (fast, default), or scrollable (TYPE_SCROLL_INSENSITIVE/SENSITIVE).
- **Concurrency**: CONCUR_READ_ONLY (default) or CONCUR_UPDATABLE.
- **Holdability**: whether cursors stay open across commits.
- **Typed getters**: getInt, getString, getBigDecimal, getObject, handling NULL via wasNull().
- **Performance**: tune fetchSize; stream large results.

Sample code

```
String sql = "SELECT id, name, salary FROM employees";
try (PreparedStatement ps = conn.prepareStatement(sql,
      ResultSet.TYPE_FORWARD_ONLY,
      ResultSet.CONCUR_READ_ONLY);
    ResultSet rs = ps.executeQuery()) {
   while (rs.next()) {
      int id = rs.getInt(1);
      String name = rs.getString("name");
      double sal = rs.getDouble("salary");
      System.out.printf("%d %s %.2f%n", id, name, sal);
   }
}
```

Real-life examples

- Generating a **PDF report** by streaming thousands of rows from ResultSet without loading all into memory.
- Using a **scrollable ResultSet** in a desktop admin tool to jump to previous/next pages.

9) What is auto-commit in JDBC?

Explanation (in depth)

- **Auto-commit = true (default)**: each SQL statement is its own transaction and commits automatically on success.
- **Auto-commit = false**: you control transactions; group multiple statements and call commit() or rollback().
- Use manual transactions for **atomic multi-step** operations and better performance (fewer commits).
- You can use **savepoints** for partial rollbacks.

Sample code

```
conn.setAutoCommit(false);
try (PreparedStatement a = conn.prepareStatement(
        "UPDATE accounts SET balance = balance - ? WHERE id = ?");
     PreparedStatement b = conn.prepareStatement(
        "UPDATE accounts SET balance = balance + ? WHERE id = ?")) {

   a.setBigDecimal(1, amount); a.setInt(2, fromId); a.executeUpdate();
   b.setBigDecimal(1, amount); b.setInt(2, toId);   b.executeUpdate();

   conn.commit(); // both updates succeed together
} catch (SQLException e) {
   conn.rollback(); // both updates undone together
} finally {
   conn.setAutoCommit(true);
}
```

Real-life examples

- **Money transfer**: debit and credit must succeed or fail as one.
- **Bulk import**: insert 1,000 rows, commit once for speed and atomicity.

10) How to connect Java to MySQL?

Explanation (in depth)

Steps:

1. **Add the driver** (MySQL Connector/J JAR or Maven/Gradle dependency).
2. **Build the JDBC URL**: jdbc:mysql://host:port/schema plus options as needed (e.g., SSL).
3. **Get a Connection** via DriverManager (simple) or a **DataSource** (preferred for pooling).
4. **Optional tuning**: character set, time zone, SSL, connection timeouts.

Sample code (DriverManager)

```
String url =
"jdbc:mysql://localhost:3306/employee_db?useSSL=false&allowPublicKey
Retrieval=true";
try (Connection conn = DriverManager.getConnection(url, "root",
"password")) {
   System.out.println("Connected!");
}
```

Sample code (DataSource + pool, recommended)

```
// Example using HikariCP (conceptual; add dependency in your build)
import com.zaxxer.hikari.*;
HikariConfig cfg = new HikariConfig();
cfg.setJdbcUrl("jdbc:mysql://localhost:3306/employee_db");
cfg.setUsername("root");
cfg.setPassword("password");
HikariDataSource ds = new HikariDataSource(cfg);

try (Connection conn = ds.getConnection()) {
   // do work
}
```

Real-life examples

- A **Spring Boot** service using a pooled DataSource to handle high traffic.
- A **VS Code** Java console app connecting via DriverManager for local testing.p