What is JDBC ?

- It’s an acronym for Java Database Connectivity.

- It's an API for connecting to relational database(DB) & perform CRUD (Create ,Retrieve , Update & Delete)operations.

- The java module used is java.sql.

- The packages used are java.sql , for core JDBC functionality & javax.sql for its extension , to be used in developing enterprise applications.

- Instead of fixed connectivity javax.sql , supports connection pooling .

Why JDBC ?

- It allows developers to build Java applications which are

- **platform independent** + **partially DB independent**

- It Continues with **WORA** (Write once run anywhere – run on any DB) design philosophy.

What makes JDBC database-independent?

1. **JDBC API (specifications)**
    - Defined in the **JDK (java.sql, javax.sql)**.
    - Contains mainly **interfaces** (e.g., Connection, Statement, ResultSet).
    - These describe **what operations** can be done, not **how** they’re implemented.
    - This API is **common across all databases** → the same code compiles regardless of which DB you use.
2. **JDBC Drivers (implementations)**
    - Provided by **database vendors** (e.g., Oracle, MySQL, PostgreSQL, Mongo JDBC bridge) or third-party providers.
    - These implement the JDBC interfaces in a way that talks to the

specific DB using its **native protocol**.

- Example:
  - com.mysql.cj.jdbc.Driver → MySQL
  - oracle.jdbc.driver.OracleDriver → Oracle
  - org.postgresql.Driver → PostgreSQL
- This **decouples your Java code from the database details**. You just need the right driver JAR at runtime.

3. **DriverManager / DataSource as mediator**
   - DriverManager.getConnection(url, user, pwd) chooses the **correct driver** based on the URL (jdbc:mysql://..., jdbc:oracle://...).
   - Your code only depends on the **JDBC API**, not on the driver classes themselves
   - This allows swapping databases with little/no code change (just change URL & driver JAR).

Flow of JDBC independence

1. Java code calls JDBC API (e.g., connection.prepareStatement("SELECT * FROM users")).
2. JDBC API forwards to the **driver's implementation** (e.g., MySQL's driver).
3. Driver **translates JDBC calls → DB native protocol** (e.g., SQL string, binary protocol).
4. DB executes query and returns raw results.
5. Driver **maps DB results → Java objects** (e.g., ResultSet).

JDBC Driver Types

JDBC drivers are the **bridge between Java and the database**. There are **4 standard types**

1 Type-1: JDBC-ODBC Bridge Driver

- **How it works:**
  Java calls → JDBC API → **ODBC driver** → Database.
- **Dependency:** Requires ODBC installed on client machine.
- **Pros:**
  - Easy to use for
  - Works with almost any DB that has an ODBC driver.
- **Cons:**
  - Slow (many layers of translation).
  - Platform specific.
  - Deprecated & removed since **Java 8**.
- **Example:** sun.jdbc.odbc.JdbcOdbcDriver

2 Type-2: Native-API Driver

- **How it works:**
  Java calls → JDBC API → **Native DB client library (C/C++)** → Database.
- **Dependency:** Requires native DB libraries installed on client.
- **Pros:**
  - Faster than Type-1 (less translation).
  - Uses native DB features.
- **Cons:**
  - Requires platform-specific native libraries.
  - Harder to deploy.
- **Example:** Oracle's **OCI (**Oracle Call Interface**) driver** (oracle.jdbc.driver.OracleDriver).

3 Type-3: Network Protocol Driver

- **How it works:**
  Java calls → JDBC API →Middleware S**erver side (JDBC Driver → DB protocol)** → Database.
- **Dependency:** Needs a middleware server between client and DB.
- **Pros:**
  - No client-side native libraries needed.
  - Can work with multiple databases (middleware does translation).
  - Easier for internet-based apps.
- **Cons:**
  - Middleware server adds an extra network layer
- **Example:** IBM WebSphere Net drivers.

4 Type-4: Thin Driver (Pure Java Driver)

- **How it works:**
  Java calls → JDBC API → **Database protocol (via TCP/IP)** → Database.
- **Dependency:** None . Pure Java.
- **Pros:**
  - Fast (direct communication with DB).
  - Platform-independent.
  - Easy deployment (just add driver JAR).
- **Cons:**
  - DB-specific (each vendor provides its own driver).
- **Example:**

- MySQL → com.mysql.cj.jdbc.Driver
- Oracle Thin Driver → oracle.jdbc.OracleDriver

Generic development Steps in JDBC n implementation.

1. In earlier versions, it was mandatory to load JDBC driver first.

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

   Not required in since JDBC 4.0 (JDK 6+)

Compatibility note – You can use safely mysql connector 9.3 jar with JDK 21 & Mysql 8+ database.

2. Connect to DB

   API (method) of java.sql.DriverManager class

   public static Connection getConnection(String dbURL,String userName,String password) throws SQLException

   Syntax for dbURL : protocol : subprotocol : db details

   Example For locally hosted My sql connector – jdbc:mysql://localhost:3306/test

   Test – DB instance name

3. Create empty (not holding the SQL yet) Statement to execute SQL , from the Connection interface.

   Use Connection interface method

   public Statement createStatement() throws SQLException

4. To Execute select query

   Method of Statement interface

   public ResultSet executeQuery(String selectQuery) throws SQLException

5. To Execute DDL|DML query
   Use **Statement** interface.

It is used to execute **static SQL queries** (typically without IN parameters.).
Used typically in case of non repetitive queries
How it works:

- SQL is sent to the DB **as-is** every time.
- The DB parses, compiles, and executes it each time you call executeQuery() or executeUpdate().

Drawbacks:

- No query caching (compiled again on each call).
- Vulnerable to **Interfaceection** if user input is concatenated into the query.

(For more details on **Interfaceection** , refer - " \Interfaceection\Regarding Interfaceection.txt")

- Slower for repeated executions with different parameters.
- Better alternative is , PreparedStatement

Method of Statement interface

public int executeUpdate(String DDLOrDML) throws SQLException

Returns number of rows updated|inserted|deleted

Returns 0 , in case of DDL

6. To Process a ResultSet in case of select query

Java.sql.ResultSet is an interface

- It represents DB result set. It internally maintains a cursor. Cursor is initially positioned before the 1st row.

Methods of ResultSet

1. public boolean **next()** throws SQLException

It tries to advance the cursor to the next row. It returns true if next row exists , otherwise false.

2. To read column data

public String getString(int columnIndex) throws SQLException

public int getInt(int columnIndex) throws SQLException

public double getDouble(int columnIndex) throws SQLException

public java.sql.Date getDate(int columnIndex) throws SQLException

etc.

Instead of column index , you can pass column name

Eg. String getString(String columnLabel) throws SQLException

7. Close (clean up) DB resources , by closing ResultSet , Statement & Connection.

In most of the real world scenarios, use PreparedStatement instead , a sub interface of the Statement interface.

PreparedStatement (extends Statement)

- It is a **precompiled SQL** statement that can accept **parameters** (placeholders ?).
- **How it works:**
    - SQL is compiled **once**, and the DB can reuse the execution plan.
    - Values are bound later using setter methods (setInt, setString, setDate etc.).

Advantages:

- **Performance**: query plan is cached by DB → faster when executed multiple times.
- **Security**: prevents **Interfaceection** (parameters are sent separately, not string-concatenated).

- **Convenience**: no need to escape quotes, handle special characters manually.

Development steps

1. Create PreparedStatement , using Connection interface method

API of Connection

public PreparedStatement prepareStatement(String sql) throws SQLException

2. Set the values of IN parameters

API of PreparedStatement

public void setType(int parameterIndex,Type value) throws SQLException

Type – JDBC data type

eg - setString , setDate,setBoolean etc.

3. To execute select query

public ResultSet executeQuery() throws SQLException

Processing of Resultset is same as earlier.

4. To execute DDL | DML

public int executeUpdate() throws SQLException

5. Close (Clean up) DB resources (close ResultSet,PreparedStatement , Connection).

For the invocation of Stored Procedure | stored function , use CallableStatment (extends PreparedStatement)

Development steps

1. Create Callable statement from Connection interface.

Use Connection interface method

public CallableStatement prepareCall(String sql) throws SQLException

sql syntax for calling a procedure - {call procedureName(?,?,?,....?)}

sql syntax for calling a function - {?=call functionName(?,?,?,....?)}

{} : Escape sequence meant for JDBC driver to translate the procedure | function invocation in DB specific manner.

? : IN | OUT | IN OUT

2. For OUT as well as IN OUT parameters ,

Register them with JVM (i.e you will have to specify generic SQL type - available from constants in java.sql.Types class , before the execution)

API of CallableStatement

public void registerOutParameter(int parameterIndex,int sqlType) throws SQLException

3. Set IN parameters

Methods inherited from PreparedStatement

public void setType(int parameterIndex,Type value);

4. Execute the procedure | function

public boolean execute() throws SQLException

Ignore return value here.

5. Read results of procedure | function from OUT parameters

Method of CallableStatement

public Type getType(int parameterIndex) throws SQLException

parameterIndex - index of OUT parameter.

type - JDBC data type (generic SQL type)

Default type of the ResultSet cursor

TYPE_FORWARD_ONLY : can traverse in forward direction only.

READ_ONLY : can only use getters to read data from Result set

For additional reading, refer - " JDBC ResultSets.txt"

For production grade application, it is always recommended to use a layered architecture.

Why Layered Architecture?

A layered architecture divides the application into **logical layers**, each with a **specific responsibility**:

1. **UI / Presentation Layer**
    - To interact with the user (request & response).
    - To keep user interaction separate from business logic & data access logic.

2. **Service / Business Layer** (optional but recommended)
    - **To** implement **business logic** (calculations, validation, business rules).
    - Makes **unit testing easier**.
    - Enables **reusing business logic** in multiple UIs (web, desktop, API).

3. **DAO / Repository Layer**
    - **DAO = Data Access Object**
    - To encapsulate DB access operations
    - Makes it easy to **switch databases** or change SQL without affecting business logic.
    - Simplifies testing (you can mock DAO for unit tests).

4. **POJO / Model / Entity Layer**
    - **POJO = Plain Old Java Object**
    - It is a simple Java class with **fields, getters, setters**, and optionally constructors.
    - Used to **r**epresent **domain objects** (e.g., User, Product, Course,Student,BankAccount).
    - **Acts as data carrier** between layers.
    - Avoids mixing DB logic, business logic, and UI fields
    - Forms the basis of ORM (Object-Relational Mapping) , map **Java objects** to **database tables**. For more details on ORM , refer the diagram - " ORM overview.png"

5. **DB Utils / Connection Layer**
    - To centralize DB connection logic, avoids duplication.
    - Makes it easier to **change DB configuration** in one place.
    - Refer to - " day1_help\DB connection singleton pattern.txt"

ORM (Object - Relational Mapping)

What is ORM?

- **O**bject **R**elational **M**apping = Technique to **map Java objects (POJOs) to database tables** automatically.
- It **bridges the gap** between the **object-oriented world (Java classes)** and the **relational world (SQL tables)**.

  Example:
- class User {

    private int id;

```
        private String name;
    }
```

↔ mapped to ↔

```
CREATE TABLE users (
    id INT PRIMARY KEY,
    name VARCHAR(100)
);
```

Why ORM?

In plain JDBC:

- You write SQL queries (INSERT, SELECT) manually.
- You handle connections, statements, and result sets.
- You need to convert SQL types ↔ Java types.

With ORM (like Hibernate/JPA):

- Just work with Java objects → framework **auto-generates SQL** and manages persistence.
- **No need** to manually handle ResultSet, type conversions, or joins.

ORM Concepts (Key Mapping)

1. **Entity Class(POJO class) ↔ Table**
    - One class = One table.
2. **Entity Field ↔ Column**
    - One property = One column.
3. Entity object(POJO) ↔ **one row**
4. Can additionally support **Associations & Inheritance**

It will be actually used from Hibernate onwards. Think of this as an introduction to the term.

JDBC Transaction overview

- A **transaction** in JDBC (or in general in DB) is a **sequence of one or more SQL statements executed as a single logical unit of work**.
- Either **all statements succeed (commit)**
- Or **all statements fail (rollback)**
- Ensures **ACID properties**:
    - **A**tomicity → all or nothing
    - **C**onsistency → DB moves from one valid state to another
    - **I**solation → concurrent transactions don't interfere
    - **D**urability → once committed, changes persist

By default, **JDBC connections are opened in auto-commit mode**, meaning:

Connection conn = DriverManager.getConnection(...);

- Each SQL statement is **committed automatically** after execution.
- No need to call commit().

3.To manage the transactions manually:

- Disable auto commit flag.
- Execute multiple related SQL statements within a try block.
- At the end of the try block(indicates success) , commit the transaction.
- In catch block , rollback the transaction. (all or nothing !)
- Connection interface methods
- Enable auto commit flag

4. Methods of Connection interface
    - void setAutoCommit (boolean false) throws SQLException

- void commit() throws SQLException
- void rollback() throws SQLException

5. A **partial rollback** allows you to **undo only part of a transaction** instead of the entire transaction. A **Savepoint** marks an intermediate point within a transaction. You can **rollback to a savepoint** without undoing the work done before it.
   Methods of Connection interface
   - To create a savepoint
     SavePoint setSavePoint() throws SQLException
   - To rollback to a savepoint
     void rollback(SavePoint savePoint) throws SQLException
   - To release a savepoint
     void releaseSavePoint(SavePoint savePoint) throws SQLException