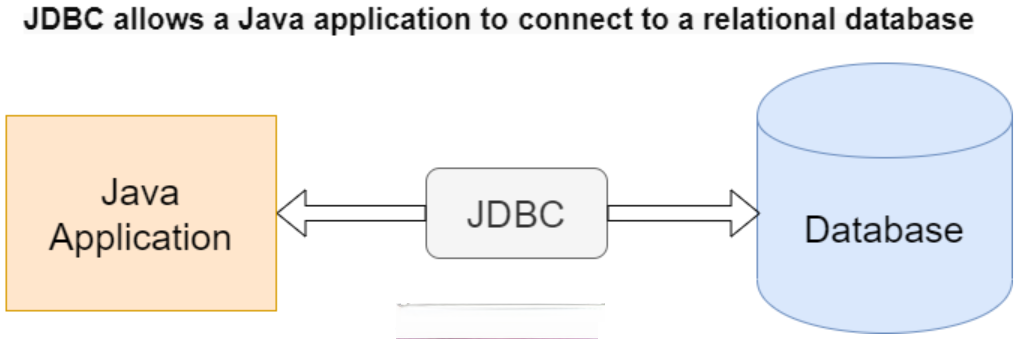
**JDBC (Java Database Connectivity)** is an API that allows Java applications to interact with relational databases.

It provides methods for querying and updating data in databases.



The JDBC API is comprised of two packages:

* [**java.sql**](https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html)
* [**javax.sql**](https://docs.oracle.com/javase/8/docs/api/javax/sql/package-summary.html)

We automatically get both packages when you download the Java Platform Standard Edition (Java SE).

**JDBC API Components**

**Key Classes and Interfaces**

**1. DriverManager**

* Manages a list of database drivers.
* Establishes a connection to the database.

**2. Connection**

* Represents a session/connection with a specific database.

**3. Statement**

* Used for executing a static SQL statement and returning the results.

**4. PreparedStatement**

* Extends Statement.
* Used for executing precompiled SQL statements with or without input parameters.

**5. CallableStatement**

* Extends PreparedStatement.
* Used to execute SQL stored procedures.

**6. ResultSet**

* Represents the result set of a database query.

**7. ResultSetMetaData**

* Provides information about the types and properties of the columns in a ResultSet.

**8. DatabaseMetaData**

* Provides comprehensive information about the database as a whole.

**JDBC Drivers**

JDBC can work with any database as long as proper drivers are provided.

A JDBC driver is a JDBC API implementation used for connecting to a particular type of database.

There are several types of JDBC drivers:

* **JDBC-ODBC Bridge Driver (Type 1)** – contains a mapping to another data access API i.e. ODBC.
* **Native-API Driver (Type 2)** – is an implementation that uses client-side libraries of the target database
* **Network Protocol Driver (Type 3)** – uses middleware to convert JDBC calls into database-specific calls
* **Thin Driver (Type 4)** – connect directly to a database by converting JDBC calls into database-specific calls

***Type 4*** *(Thin Driver) is the most commonly used, due to its portability and high performance.*

**Connecting to a Database**

**Steps for JDBC Connection and Executing Statements**

1. **Load the JDBC Driver**: The JDBC driver is needed to communicate with the database.
2. **Establish a Connection**: Use DriverManager to get a connection to the database.
3. **Create a Statement**: Use the connection to create a statement.
4. **Execute a Query**: Execute the SQL queries (select, update, etc.).
5. **Process the Result Set**: Retrieve data from the query and process it.
6. **Close Connections**: Always close the connection, statement, and result set to avoid resource leakage.

**Step 1: Load and Register the Driver**

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

*For JDBC 4.0 and newer, the driver is automatically loaded when the getConnection() method is called*

**Step 2: Establish a Connection**

Use the DriverManager class to establish a connection to the database:

**String url = "jdbc:mysql://localhost:3306/mydatabase";**

**String user = "root";**

**String password = "password";**

**Connection connection = DriverManager.getConnection(url, user, password);**

**Step 3: Create a Statement**

Create a Statement object to execute SQL queries:

**Statement statement = connection.createStatement();**

**Step 4: Execute SQL Queries**

**ResultSet resultSet = statement.executeQuery("SELECT \* FROM employees");**

**Step 5: Process the Results**

Process the ResultSet obtained from the query:

**while (resultSet.next()) {**

**int id = resultSet.getInt("id");**

**String name = resultSet.getString("name");**

**String email = resultSet.getString("email");**

**System.out.println(id + ", " + name + ", " + email);**

**}**

**Step 7: Close the Connections**

Finally, close the ResultSet, Statement, and Connection objects to release the resources:

**resultSet.close();**

**statement.close();**

**connection.close();**

**What is ResultSet?**

The ResultSet interface represents the result set of a database query.

It provides methods to iterate through the results, retrieve individual columns by name or index, and update the data.

The ResultSet maintains a cursor pointing to its current row of data, and the cursor can be moved forward, backward, and to specific rows.

**// Create a Statement object to execute the SQL query**

Statement statement = connection.createStatement();

**// Execute a query (SELECT statement)**

String query = "SELECT id, name, email FROM users";

ResultSet resultSet = statement.executeQuery(query);

**// Process the ResultSet to extract data**

while (resultSet.next()) {

int id = resultSet.getInt("id"); // Get the 'id' column value

String name = resultSet.getString("name"); // Get the 'name' column value

String email = resultSet.getString("email"); // Get the 'email' column value

**// Print the values retrieved**

System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);

}

**Types of statement objects**

* 1. Statement
  2. PreparedStatement
  3. CallableStatement

**PreparedStatement:**

The PreparedStatement interface is a sub interface of Statement.

An SQL statement is precompiled and stored in a PreparedStatement object, which can then be used to efficiently execute it multiple times.

**SQL Query Execution**

**Statement**:

If you're executing the same query repeatedly with different values, the query will be parsed and compiled every time it is run.

**Example:**

Statement stmt = connection.createStatement();

stmt.executeQuery("SELECT \* FROM users WHERE name = 'John'");

**Performance**

Can be less efficient when executing the same query multiple times with different values. Every time the query is executed, it needs to be parsed and compiled.

**PreparedStatement**:

Executes precompiled SQL queries with placeholders (?) for parameters. The query is parsed and compiled once when it is created, and you can reuse it with different parameter values.

**Example:**

PreparedStatement ps =

connection.prepareStatement("SELECT \* FROM users WHERE name = ?");

ps.setString(1, "John");

ResultSet rs = ps.executeQuery();

**Performance**

More efficient when executing the same query repeatedly with different values because the query is precompiled once and can be reused. This can lead to better performance, especially in cases of multiple executions.

**CallableStatement:**

The **CallableStatement** interface is a sub interface of Statement.

Used to execute SQL stored procedures.

Stored procedures are beneficial for encapsulating business logic and improving performance.

Stored procedures are precompiled SQL statements stored in the database.

**Setting Up the MySQL Database**

|  |
| --- |
| CREATE TABLE products (  id INT AUTO\_INCREMENT PRIMARY KEY,  name VARCHAR(100),  description VARCHAR(255),  price DECIMAL(10, 2)  );  INSERT INTO products (name, description, price) VALUES  ('Laptop', 'Dell Inspiron', 75000.00),  ('Smartphone', 'Samsung Galaxy', 30000.00),  ('Tablet', 'Apple iPad', 50000.00); |

**Creating Stored Procedures in MySQL**

Create a stored procedure that will insert a new product into the products table.

|  |
| --- |
| DELIMITER //  CREATE PROCEDURE addProduct (  IN p\_name VARCHAR(100),  IN p\_description VARCHAR(255),  IN p\_price DECIMAL(10, 2)  )  BEGIN  INSERT INTO products (name, description, price) VALUES (p\_name, p\_description, p\_price);  END //  DELIMITER ; |

**Executing Stored Procedures with CallableStatement**

|  |
| --- |
| String callSQL = "{CALL addProduct(?, ?, ?)}";  try (Connection connection = JDBCExample.getConnection();  CallableStatement callableStatement = connection.prepareCall(callSQL)) {  callableStatement.setString(1, "Smartwatch");  callableStatement.setString(2, "Apple Watch Series 6");  callableStatement.setBigDecimal(3, new Big Decimal("40000.00"));  callableStatement.execute();  System.out.println("Stored procedure executed successfully.");  } catch (SQLException e) {  e.printStackTrace();  } |

**Retrieving Output Parameters from CallableStatement**

Create another stored procedure that retrieves the price of a product and returns it as an output parameter.

|  |
| --- |
| DELIMITER //  CREATE PROCEDURE getProductPrice (  IN p\_name VARCHAR(100),  OUT p\_price DECIMAL(10, 2)  )  BEGIN  SELECT price INTO p\_price FROM products WHERE name = p\_name;  END //  DELIMITER ; |

**Using CallableStatement call this stored procedure and retrieve the output parameter.**

|  |
| --- |
| String callSQL = "{CALL getProductPrice(?, ?)}";  try (Connection connection = JDBCExample.getConnection();  CallableStatement callableStatement = connection.prepareCall(callSQL)) {  callableStatement.setString(1, "Laptop");  callableStatement.registerOutParameter(2, java.sql.Types.DECIMAL);  callableStatement.execute();  BigDecimal price = callableStatement.getBigDecimal(2);  System.out.println("Price of Laptop: " + price);  } catch (SQLException e) {  e.printStackTrace();  } |

**What is ResultSetMetaData:**

The ResultSetMetaData interface is part of the JDBC API.

It provides information about the types and properties of the columns in a ResultSet object.

This metadata can be used to determine the number of columns, their types, names, and other attributes.

**Using ResultSetMetaData to Get Column Information**

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery("SELECT \* FROM products"))

ResultSetMetaData metaData = resultSet.getMetaData();

int columnCount = metaData.getColumnCount();

System.out.println("Number of Columns: " + columnCount);

for (int i = 1; i <= columnCount; i++) {

String columnName = metaData.getColumnName(i);

String columnTypeName = metaData.getColumnTypeName(i);

int columnDisplaySize = metaData.getColumnDisplaySize(i);

boolean isNullable = metaData.isNullable(i) = ResultSetMetaData.columnNullable;

boolean isAutoIncrement = metaData.isAutoIncrement(i);

System.out.println("Column Name: " + columnName);

System.out.println("Column Type: " + columnTypeName);

System.out.println("Column Size: " + columnDisplaySize);

System.out.println("Is Nullable: " + isNullable);

System.out.println("Is Auto Increment: " + isAutoIncrement);

System.out.println("---------------------------");

}

// Retrieve data from ResultSet

while (resultSet.next()) {

int id = resultSet.getInt("id");

String name = resultSet.getString("name");

String description = resultSet.getString("description");

double price = resultSet.getDouble("price");

System.out.println("ID: " + id + ", Name: " + name + ", Description: " + description + ", Price: " + price);

}

**What is DatabaseMetaData?**

The *DatabaseMetadata* interface can be used to obtain general information about the database such as the tables, stored procedures, or SQL dialect.

CREATE TABLE users (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

email VARCHAR(100),

country VARCHAR(100),

password VARCHAR(100)

);

**Retrieving General Database Information**

We will use the DatabaseMetaData interface to retrieve general information about the database.

DatabaseMetaData metaData = connection.getMetaData();

// Retrieve database information

String databaseProductName = metaData.getDatabaseProductName();

String databaseProductVersion = metaData.getDatabaseProductVersion();

String driverName = metaData.getDriverName();

String driverVersion = metaData.getDriverVersion();

String url = metaData.getURL();

String userName = metaData.getUserName();

System.out.println("Database Product Name: " + databaseProductName);

System.out.println("Database Product Version: " + databaseProductVersion);

System.out.println("Driver Name: " + driverName);

System.out.println("Driver Version: " + driverVersion);

System.out.println("URL: " + url);

System.out.println("User Name: " + userName);

**Retrieving Tables Information**

**We will use the DatabaseMetaData interface to retrieve information about the tables in the database.**

DatabaseMetaData metaData = connection.getMetaData();

// Retrieve tables information

ResultSet tables = metaData.getTables(null, null, "%", new String[] {"TABLE"});

System.out.println("Tables:");

while (tables.next()) {

String tableName = tables.getString("TABLE\_NAME");

String tableType = tables.getString("TABLE\_TYPE");

System.out.println("Table Name: " + tableName + ", Table Type: " + tableType);

}

**Retrieving Columns Information**

We will use the DatabaseMetaData interface to retrieve information about the columns in the users table.

DatabaseMetaData metaData = connection.getMetaData();

// Retrieve columns information

ResultSet columns = metaData.getColumns(null, null, "users", "%");

System.out.println("Columns in 'users' table:");

while (columns.next()) {

String columnName = columns.getString("COLUMN\_NAME");

String columnType = columns.getString("TYPE\_NAME");

int columnSize = columns.getInt("COLUMN\_SIZE");

boolean isNullable = columns.getInt("NULLABLE") == DatabaseMetaData.columnNullable;

System.out.println("Column Name: " + columnName + ", Column Type: " + columnType +

", Column Size: " + columnSize + ", Is Nullable: " + isNullable);

}

**Handling Transactions**

**What is a Transaction?**

A transaction is a sequence of operations performed as a single logical unit of work. A transaction ensures the ACID properties: Atomicity, Consistency, Isolation, and Durability.

* **Atomicity:** Ensures that all operations within the transaction are completed successfully. If any operation fails, the entire transaction is rolled back.
* **Consistency:** Ensures that the database is in a consistent state before and after the transaction.
* **Isolation:** Ensures that transactions are isolated from each other until they are completed.
* **Durability:** Ensures that the changes made by the transaction are permanent and stored in the database.

**Why Use Transactions?**

Transactions are essential for maintaining data integrity and consistency, especially when performing multiple operations that depend on each other. Without transactions, partial updates can lead to data inconsistencies and corruption.

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

**Managing Transactions in JDBC**

In JDBC, transactions are managed using the Connection object. By default, JDBC connections are in auto-commit mode, meaning each SQL statement is committed to the database as soon as it is executed.

To manage transactions manually, you need to:

1. Disable auto-commit mode.
2. Perform your operations.
3. Commit the transaction if all operations are successful.
4. Roll back the transaction if any operation fails.

**JDBC Transaction Example**

In this example, we will transfer money between two employees by updating their salaries within a transaction.

Connection connection = null;

try {

connection.setAutoCommit(false); // Disable auto-commit mode

// Prepare SQL statements

String withdrawSQL = "UPDATE employee SET salary = salary - ? WHERE name = ?";

String depositSQL = "UPDATE employee SET salary = salary + ? WHERE name = ?";

try (PreparedStatement withdrawStmt = connection.prepareStatement(withdrawSQL);

PreparedStatement depositStmt = connection.prepareStatement(depositSQL)) {

// Withdraw money from John Doe

withdrawStmt.setBigDecimal(1, new BigDecimal("5000.00"));

withdrawStmt.setString(2, "John Doe");

withdrawStmt.executeUpdate();

// Simulate an error

// int x = 1 / 0;

// Deposit money to Jane Smith

depositStmt.setBigDecimal(1, new BigDecimal("5000.00"));

depositStmt.setString(2, "Jane Smith");

depositStmt.executeUpdate();

// Commit the transaction

connection.commit();

System.out.println("Transaction committed successfully.");

} catch (SQLException ex) {

// Roll back the transaction in case of an error

if (connection != null) {

try {

connection.rollback();

System.out.println("Transaction rolled back.");

} catch (SQLException e) {

e.printStackTrace();

}

}

ex.printStackTrace();

}

} catch (SQLException e) {

e.printStackTrace();

}