JDBC:

JDBC (Java Database Connectivity) is a way for Java programs to connect and interact with databases, allowing software to store, retrieve, and manage data easily. It uses a simple, step-by-step process to connect Java applications to many types of databases using special drivers and SQL queries.

What is JDBC and Why is Data Important?

Data is central to all software and technology, driving applications like messaging and banking. Databases store this data, and types include flat files, RDBMS (like MySQL, Oracle), NoSQL (like MongoDB), and big data systems.

Connecting Java Applications to Databases

Java applications interact with databases by sending SQL queries. Users typically interact with a graphical interface, which communicates with the backend database through Java code.

What is JDBC?

JDBC stands for Java Database Connectivity and is the standard way to connect Java applications to databases. It requires a driver, which acts as a bridge between Java and the database.

Types of JDBC Drivers

There are four types of JDBC drivers: ODBC Bridge, Native API, Network Protocol, and Pure Java drivers. Each type is suited for different systems and requirements.

The Seven Steps of JDBC Connectivity

The JDBC process involves seven steps: 1) Import the package, 2) Load and register the driver, 3) Establish the connection, 4) Create a statement, 5) Execute the query, 6) Process results, and 7) Close the connection and statement.

Step 1: Importing the JDBC Package

The required package is java.sql.*, which provides all necessary classes and interfaces for JDBC operations.

Step 2: Loading and Registering the Driver

The driver must be loaded (usually as a .jar file) and registered using the Class.forName() method. The driver type depends on the database being used (e.g., MySQL, PostgreSQL).

Step 3: Establishing the Connection

A connection is created using the DriverManager.getConnection() method, which needs the database URL, username, and password.

Step 4: Creating Statement Objects

- For Normal statements → Use method String.format() for query.
- For Prepared statements → No need to use.

🔑 **Why difference?**

- **Statement**: SQL = plain string → you must build it yourself (String.format, + concatenation). SQL injection prone.
- **PreparedStatement**: SQL = precompiled template with placeholders → you only supply values, no need for formatting. Not prone to SQL injection.

👉 In short:

- We use String.format() in **normal statements** to manually insert values into the SQL string.
- We don’t use it in **prepared statements** because the JDBC API already provides methods (setString, setInt, etc.) to safely bind

values to placeholders.

Statements allow SQL queries to be sent to the database. Types include Statement, PreparedStatement (for parameterized queries), and CallableStatement (for stored procedures).

Step 5: Executing Queries

SQL queries are executed using methods like executeQuery() for retrieving data or executeUpdate() for modifying data. The response depends on the query type (data table or affected rows).

Step 6: Processing Results

Results are stored in a ResultSet object, which can be navigated row by row using rs.next(). Data can be retrieved by column using methods like getInt() or getString().

Step 7: Closing Connections and Statements

It is important to close Statement and Connection objects with close() methods to free up resources and

avoid database issues.

SQL Query Types in JDBC

SQL commands are grouped as DDL (Data Definition Language), DML (Data Manipulation Language), and DQL (Data Query Language). Different JDBC methods are used for each: executeQuery() for DQL, executeUpdate() for DML.

Code for Retrieval from Database:

```
import java.sql.*;

public class Main {  new *
    // For connection
    private static final String url = "jdbc:mysql://127.0.0.1:3306/mydb";  1 usage

    private static final String username = "root";  1 usage

    private static final String password = "root123";  1 usage

    public static void main(String[] args) {  new *
        // Driver Loaded
        try{
            Class.forName( className: "com.mysql.cj.jdbc.Driver");
        }
        catch(ClassNotFoundException e){
            System.out.println(e.getMessage());
        }
```

```java
// Create Connection to Database
try{
    Connection connection = DriverManager.getConnection(url, username, password);
    Statement  statement =  connection.createStatement();
    String query = String.format("select * from students");

    ResultSet resultSet = statement.executeQuery(query);
    while(resultSet.next()){
        int id = resultSet.getInt( columnLabel: "id");
        String name = resultSet.getString( columnLabel: "name");
        int age = resultSet.getInt( columnLabel: "age");
        double marks = resultSet.getDouble( columnLabel: "marks");
        System.out.println("ID: " + id);
        System.out.println("NAME: " + name);
        System.out.println("AGE: " + age);
        System.out.println("MARKS: " + marks);
    }
}
catch(SQLException e){
    System.out.println(e.getMessage());
}
```

Code for Insertion in Database:

```java
import java.sql.*;

public class Main {  new *
    // For connection
    private static final String url = "jdbc:mysql://127.0.0.1:3306/mydb";  1 usage

    private static final String username = "root";  1 usage

    private static final String password = "root123";  1 usage

    public static void main(String[] args) {  new *
        // Driver Loaded
        try{
            Class.forName( className: "com.mysql.cj.jdbc.Driver");
        }
        catch(ClassNotFoundException e){
            System.out.println(e.getMessage());
        }
```

```java
        // Create Connection to Database
        try{
            Connection connection = DriverManager.getConnection(url, username, password);
            Statement  statement =  connection.createStatement();
            String query = String.format("UPDATE students SET marks = %f WHERE id = %d", 89.5, 2);

            int rowsAffected = statement.executeUpdate(query);
            if(rowsAffected > 0){
                System.out.println("Data Updated Successfully!");
            }
            else{
                System.out.println("Data not Updated!");
            }
        }
        catch(SQLException e){
            System.out.println(e.getMessage());
        }

    }
}
```

Code for Optimized (Prepared Statement) Insertion in Database:

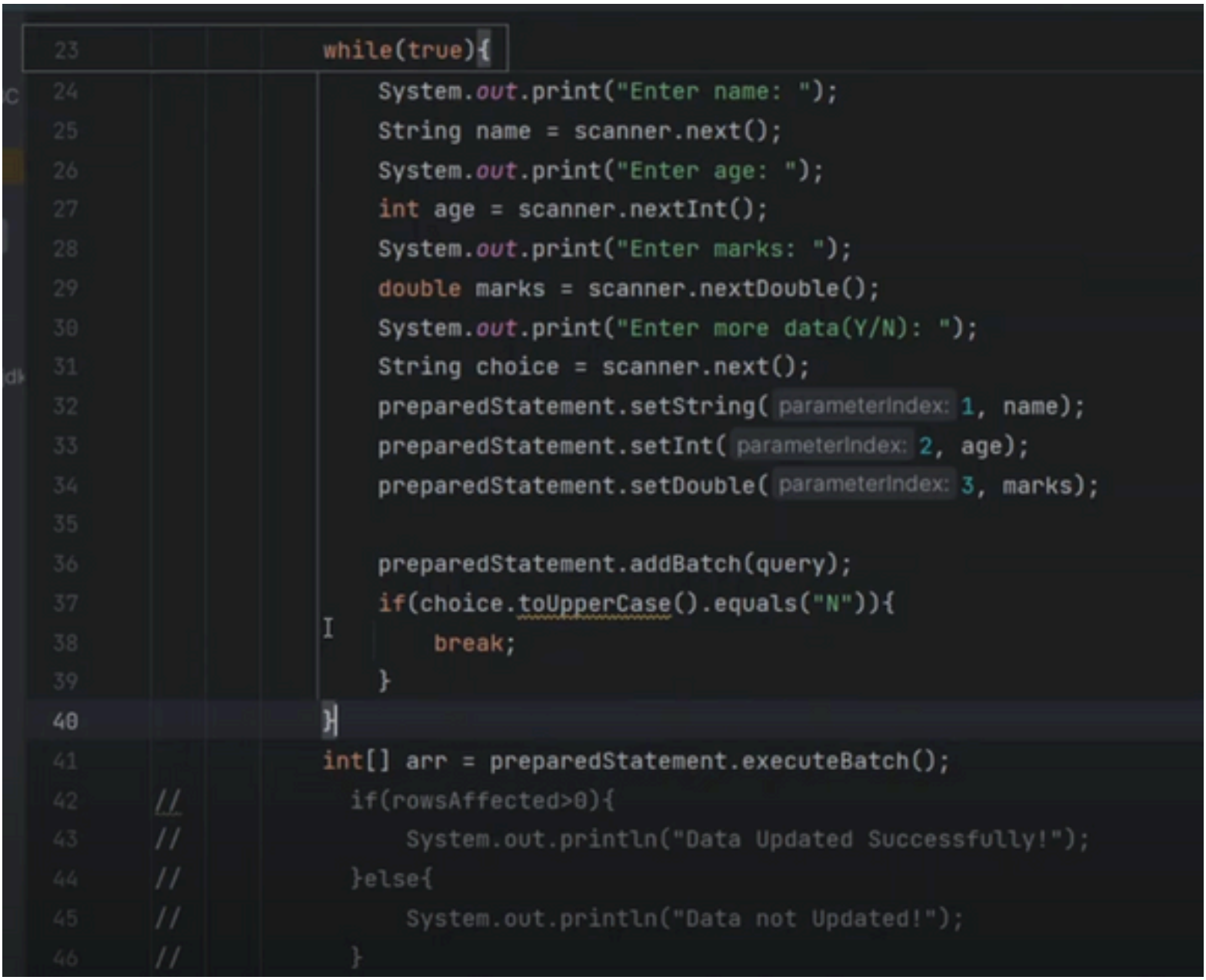
```java
public class Main {  new *
    public static void main(String[] args) {  new *

        // Create Connection to Database
        try{
            Connection connection = DriverManager.getConnection(url, username, password);
            //Statement  statement =  connection.createStatement();
            String query = "INSERT INTO students(name, age, marks) VALUES(?, ?, ?)";
            PreparedStatement preparedStatement = connection.prepareStatement(query);
            preparedStatement.setString( parameterIndex: 1,  x: "Ankita");
            preparedStatement.setInt( parameterIndex: 2,  x: 25);
            preparedStatement.setDouble( parameterIndex: 3,  x: 84.7);

            int rowsAffected = preparedStatement.executeUpdate();
            if(rowsAffected > 0){
                System.out.println("Data Inserted Successfully!!");
            }
            else{
                System.out.println("Data Not Inserted!!");
            }

        }
        catch(SQLException e){
            System.out.println(e.getMessage());
        }
```

Code for Deletion from Database:

Same as insertion but with delete query!!!!

Batch Processing:

- Multiple queries prepared in a batch and in last, the batch is executed.



```
23              while(true){
24                  System.out.print("Enter name: ");
25                  String name = scanner.next();
26                  System.out.print("Enter age: ");
27                  int age = scanner.nextInt();
28                  System.out.print("Enter marks: ");
29                  double marks = scanner.nextDouble();
30                  System.out.print("Enter more data(Y/N): ");
31                  String choice = scanner.next();
32                  preparedStatement.setString( parameterIndex: 1, name);
33                  preparedStatement.setInt( parameterIndex: 2, age);
34                  preparedStatement.setDouble( parameterIndex: 3, marks);
35
36                  preparedStatement.addBatch(query);
37                  if(choice.toUpperCase().equals("N")){
38                      break;
39                  }
40              }
41              int[] arr = preparedStatement.executeBatch();
42      //          if(rowsAffected>0){
43      //              System.out.println("Data Updated Successfully!");
44      //          }else{
45      //              System.out.println("Data not Updated!");
46      //          }
```

Transaction Handling:

- **Commit() and Rollback() methods to achieve ATOMICITY(complete or none). There should be a equilibrium.**
- **Data Consistency is achieved through above two methods.**

- By default the connection is on autocommit() == true, it does not care if transaction is complete or not.
- But in case of Transaction handling we need to control this behavior.

```
11
12      public static void main(String[] args) {  new *
13          // Driver Loaded
14          try{
15              Class.forName( className: "com.mysql.cj.jdbc.Driver");
16          }
17          catch(ClassNotFoundException e){
18              System.out.println(e.getMessage());
19          }
20
21          // Create Connection to Database
22          try{
23              Connection connection = DriverManager.getConnection(url, username, password);
24              connection.setAutoCommit(false);
25              String debit_query = "UPDATE accounts SET balance = balance - ? WHERE account_number = ?";
26              String credit_query = "UPDATE accounts SET balance = balance + ? WHERE account_number = ?";
27              PreparedStatement debitPreparedStatement = connection.prepareStatement(debit_query);
28              PreparedStatement creditPreparedStatement = connection.prepareStatement(credit_query);
29
```

```
Scanner sc = new Scanner(System.in);
System.out.println("Enter Account number (Debit): ");
int account_number = sc.nextInt();
System.out.println("Enter Amount: ");
double amount = sc.nextDouble();
System.out.println("Enter Account number (Credit): ");
int account_number2 = sc.nextInt();

debitPreparedStatement.setDouble( parameterIndex: 1, amount);
debitPreparedStatement.setInt( parameterIndex: 2, account_number);
creditPreparedStatement.setDouble( parameterIndex: 1, amount);
creditPreparedStatement.setInt( parameterIndex: 2, account_number2);

debitPreparedStatement.executeUpdate();
creditPreparedStatement.executeUpdate();
```

```
    if(isSufficient(connection, account_number, amount)){
        connection.commit();
        System.out.println("Transaction Successful!!");
    }
    else{
        connection.rollback();
        System.out.println("Transaction Failed!!");
    }
}
catch(SQLException e){
    System.out.println(e.getMessage());
}
```

```
@   static boolean isSufficient(Connection connection, int account_number, double amount){
        try{
            String query = "SELECT balance FROM accounts WHERE account_number = ?";
            PreparedStatement preparedStatement = connection.prepareStatement(query);
            preparedStatement.setInt( parameterIndex: 1, account_number);
            ResultSet resultSet = preparedStatement.executeQuery();

            if(resultSet.next()){
                double current_balance = resultSet.getDouble( columnLabel: "balance");
                if(amount > current_balance){
                    return false;
                }
            }
        }
        catch(SQLException e){
            System.out.println(e.getMessage());
        }
        return true;
    }
```