PROGRAMMING IN JAVA

Lecture 15

JDBC, SQL, and Database-Driven Java Applications

Connections, Prepared Statements, Result Sets, CRUD, Transactions, Batch Processing, DAO Design, and Integrated Architecture

Lecturer: Arshyn Altybay

Department of Computer Engineering,
Almaty Technological University, Almaty, Kazakhstan

1. Learning Objectives

After completing this lecture, students should be able to:

1. explain the role of relational databases in Java applications;
2. explain the purpose of JDBC;
3. describe the JDBC architecture;
4. explain the role of JDBC drivers;
5. establish a database connection;
6. use `Connection`, `Statement`, and `PreparedStatement`;
7. execute `SELECT`, `INSERT`, `UPDATE`, and `DELETE`;
8. process query results with `ResultSet`;
9. use try-with-resources for JDBC resources;
10. explain why `PreparedStatement` is preferred for user data;
11. explain SQL injection and how to reduce its risk;
12. use transactions with `commit()` and `rollback()`;
13. use batch operations;
14. retrieve generated keys;
15. organize persistence code using DAO or repository patterns;
16. describe connection pooling;
17. design an integrated database-driven Java application.

2. Why Databases Are Used

Files are useful for simple persistence, but many applications require more powerful data management.

A relational database provides:

- structured tables;
- relationships between data;
- searching and filtering;
- transactions;
- concurrent access;
- indexing;
- integrity constraints;
- centralized persistence.

Typical Java applications may store students, courses, employees, customers, products, orders, and payments.

3. Relational Database Concepts

A relational database organizes information into tables.

Illustration

```
STUDENT
+----+----------+-----+
| id | name     | gpa |
+----+----------+-----+
|  1 | Aruzhan  | 3.9 |
|  2 | Dias     | 3.5 |
+----+----------+-----+
```

Important concepts include:

- table;
- row;
- column;
- primary key;
- foreign key;
- constraint;

- transaction.

4. SQL

SQL stands for **Structured Query Language**.

Common operations include:

```
SELECT * FROM student;

INSERT INTO student(name, gpa)
VALUES ('Aruzhan', 3.9);

UPDATE student
SET gpa = 4.0
WHERE id = 1;

DELETE FROM student
WHERE id = 1;
```

CRUD means:

- Create;
- Read;
- Update;
- Delete.

5. What Is JDBC?

JDBC stands for Java Database Connectivity.

It is the standard Java API for communicating with relational databases.

Important JDBC types are found mainly in:

Illustration

```
java.sql
```

Important interfaces and classes include:

- `DriverManager`
- `Connection`
- `Statement`
- `PreparedStatement`
- `CallableStatement`
- `ResultSet`
- `SQLException`
- `DatabaseMetaData`

6. JDBC Architecture

Conceptually:

Illustration

```
Java Application
       |
      JDBC
       |
JDBC Driver
       |
Database Server
```

The application uses standard JDBC interfaces, while the database-specific driver translates JDBC calls into the protocol understood by the database.

7. JDBC Drivers

A JDBC driver is normally supplied by the database vendor or a trusted driver project.

Examples include drivers for PostgreSQL, MySQL, MariaDB, Oracle Database, Microsoft SQL Server, SQLite through third-party drivers, and H2.

The driver must normally be available on the application's class path or module path.

Modern JDBC drivers are commonly discovered automatically.

8. JDBC URL

A JDBC URL identifies the database connection target.

General form:

Illustration

```
jdbc:subprotocol:database-information
```

Examples:

Illustration

```
jdbc:postgresql://localhost:5432/university
jdbc:mysql://localhost:3306/university
```

The exact URL syntax depends on the JDBC driver.

9. Opening a Connection

The common approach is:

```
Connection connection =
    DriverManager.getConnection(
        url,
        username,
        password
    );
```

A database connection is an expensive resource and should be closed properly.

Try-with-resources is recommended.

Example 1. Opening a Connection

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Main {

    private static final String URL =
        "jdbc:postgresql://localhost:5432/university";

    private static final String USER =
        "postgres";

    private static final String PASSWORD =
        "secret";

    public static void main(String[] args) {

        try (Connection connection =
                DriverManager.getConnection(
                    URL,
                    USER,
                    PASSWORD)) {

            System.out.println(
                "Connected: " +
                !connection.isClosed()
            );

        } catch (SQLException e) {
```

Example 1 (continued)

```
30
31              System.out.println(
32                  "Connection failed: " +
33                  e.getMessage()
34              );
35          }
36      }
37  }
```

This example assumes a running database, a suitable JDBC driver, and valid connection settings.

10. Creating a Table

Example SQL:

```sql
CREATE TABLE student (
    id INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INTEGER NOT NULL,
    gpa DECIMAL(3,2) NOT NULL
);
```

Identity-column syntax may differ slightly between database systems.

11. Statement

A `Statement` can execute SQL text.

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

statement.executeUpdate(sql);
```

Important methods include:

- `executeQuery()`
- `executeUpdate()`
- `execute()`

Example 2. Executing a Simple Statement

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.SQLException;
4  import java.sql.Statement;
5
6  public class Main {
7
8      public static void main(String[] args) {
9
10         String url =
11             "jdbc:postgresql://localhost:5432/university";
12
13         String sql =
14             "CREATE TABLE IF NOT EXISTS course (" +
15             "id INTEGER PRIMARY KEY, " +
16             "title VARCHAR(100) NOT NULL)";
17
18         try (Connection connection =
19                 DriverManager.getConnection(
20                     url,
21                     "postgres",
22                     "secret");
23
24              Statement statement =
25                 connection.createStatement()) {
26
27             statement.executeUpdate(sql);
28
```

Example 2 (continued)

```
            System.out.println(
                "Table ready."
            );

        } catch (SQLException e) {

            e.printStackTrace();
        }
    }
}
```

12. Why PreparedStatement Is Important

A `PreparedStatement` represents SQL with parameter placeholders.

```
INSERT INTO student(name, age, gpa)
VALUES (?, ?, ?)
```

Values are supplied separately.

This improves safety, clarity, type handling, reuse, and resistance to SQL injection.

Example 3. INSERT with `PreparedStatement`

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class Main {

    public static void main(String[] args) {

        String url =
            "jdbc:postgresql://localhost:5432/university";

        String sql =
            "INSERT INTO student(name, age, gpa) " +
            "VALUES (?, ?, ?)";

        try (Connection connection =
                DriverManager.getConnection(
```

Example 3 (continued)

```
                    url,
                    "postgres",
                    "secret");

            PreparedStatement statement =
                connection
                    .prepareStatement(sql)) {

            statement.setString(
                1,
                "Aruzhan"
            );

            statement.setInt(
                2,
                19
            );

            statement.setDouble(
                3,
                3.9
            );

            int rows =
                statement.executeUpdate();

            System.out.println(
                "Inserted rows: " +
                rows
            );

        } catch (SQLException e) {

            e.printStackTrace();
        }
    }
}
```

13. Parameter Indexing

JDBC parameter positions start from 1, not zero.

```
statement.setString(1, name);
statement.setInt(2, age);
statement.setDouble(3, gpa);
```

14. Reading Data with `ResultSet`

A `SELECT` query normally returns a `ResultSet`.

```
ResultSet result =
        statement.executeQuery();
```

The cursor initially appears before the first row.

Calling:

```
result.next()
```

moves to the next row and returns whether a row exists.

Example 4. SELECT with `ResultSet`

```
import java.sql.*;

public class Main {

    public static void main(String[] args) {

        String url =
            "jdbc:postgresql://localhost:5432/university";

        String sql =
            "SELECT id, name, age, gpa " +
            "FROM student ORDER BY id";

        try (Connection connection =
                DriverManager.getConnection(
                    url,
```

Example 4 (continued)

```
                    "postgres",
                    "secret");

            PreparedStatement statement =
                connection
                    .prepareStatement(sql);

            ResultSet result =
                statement.executeQuery()) {

            while (result.next()) {

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

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

                int age =
                    result.getInt("age");

                double gpa =
                    result.getDouble("gpa");

                System.out.println(
                    id + " " +
                    name + " " +
                    age + " " +
                    gpa
                );
            }

        } catch (SQLException e) {

            e.printStackTrace();
        }
    }
}
```

15. Column Access

Result-set columns can often be read by column label or column index.

```
result.getString("name");
```

or:

```
result.getString(2);
```

Column labels are usually clearer and less fragile.

16. Mapping Rows to Objects

Database rows should often be converted into domain objects.

```
Student student =
    new Student(
        result.getInt("id"),
        result.getString("name"),
        result.getInt("age"),
        result.getDouble("gpa")
    );
```

This keeps SQL details separate from the rest of the application.

Example 5. Mapping a ResultSet Row

```
static Student mapStudent(
        ResultSet result)
        throws SQLException {

    return new Student(
        result.getInt("id"),
        result.getString("name"),
        result.getInt("age"),
        result.getDouble("gpa")
    );
}
```

17. UPDATE

Example SQL:

```sql
UPDATE student
SET name = ?, age = ?, gpa = ?
WHERE id = ?
```

Example 6. Updating a Student

```java
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class StudentRepository {

    private final Connection connection;

    public StudentRepository(
            Connection connection) {

        this.connection = connection;
    }

    public boolean updateGpa(
            int id,
            double gpa)
            throws SQLException {

        String sql =
            "UPDATE student " +
            "SET gpa = ? " +
            "WHERE id = ?";

        try (PreparedStatement statement =
                connection
                    .prepareStatement(sql)) {

            statement.setDouble(
                1,
                gpa
            );

            statement.setInt(
```

Example 6 (continued)

```
35                  2,
36                  id
37              );
38
39              return statement
40                      .executeUpdate()
41                      == 1;
42          }
43      }
44  }
```

18. DELETE

Example SQL:

```
DELETE FROM student
WHERE id = ?
```

Example 7. Deleting a Student

```
public boolean deleteById(
        int id)
        throws SQLException {

    String sql =
        "DELETE FROM student " +
        "WHERE id = ?";

    try (PreparedStatement statement =
            connection
                .prepareStatement(sql)) {

        statement.setInt(
            1,
            id
        );

        return statement
                .executeUpdate()
                == 1;
    }
}
```

19. CRUD Summary

CRUD	SQL	Typical JDBC Method
Create	`INSERT`	`executeUpdate()`
Read	`SELECT`	`executeQuery()`
Update	`UPDATE`	`executeUpdate()`
Delete	`DELETE`	`executeUpdate()`

20. SQL Injection

SQL injection can occur when untrusted input is concatenated directly into SQL.

Dangerous:

```
String sql =
    "SELECT * FROM users " +
    "WHERE username = '" +
    username + "'";
```

A malicious value could change the meaning of the SQL statement.

21. Use Parameters Instead of Concatenation

Preferred:

```
String sql =
    "SELECT * FROM users " +
    "WHERE username = ?";

PreparedStatement statement =
    connection
        .prepareStatement(sql);

statement.setString(
    1,
    username
);
```

The user input is treated as a value rather than SQL syntax.

Example 8. Safe Login Query Structure

```
public boolean userExists(
        String username)
        throws SQLException {

    String sql =
        "SELECT 1 FROM users " +
        "WHERE username = ?";

    try (PreparedStatement statement =
            connection
                .prepareStatement(sql)) {

        statement.setString(
            1,
            username
        );

        try (ResultSet result =
                statement
                    .executeQuery()) {

            return result.next();
        }
    }
}
```

Authentication systems also require secure password hashing. Plain-text password storage should not be used.

22. Generated Keys

Databases often generate primary-key values automatically.

JDBC can request generated keys.

Example 9. Reading a Generated ID

```
import java.sql.*;

public class StudentRepository {

    private final Connection connection;

    public StudentRepository(
            Connection connection) {

        this.connection = connection;
    }

    public int insert(
            String name,
            int age,
            double gpa)
            throws SQLException {

        String sql =
            "INSERT INTO student(" +
            "name, age, gpa) " +
            "VALUES (?, ?, ?)";

        try (PreparedStatement statement =
                connection.prepareStatement(
                    sql,
                    Statement
                        .RETURN_GENERATED_KEYS
                )) {

            statement.setString(
                1,
                name
            );

            statement.setInt(
                2,
                age
            );

            statement.setDouble(
                3,
                gpa
            );
```

Example 9 (continued)

```
45
46              statement.executeUpdate();
47
48              try (ResultSet keys =
49                       statement
50                           .getGeneratedKeys()) {
51
52                  if (keys.next()) {
53                      return keys.getInt(1);
54                  }
55
56                  throw new SQLException(
57                      "No generated key returned."
58                  );
59              }
60          }
61      }
62  }
```

23. Transactions

A **transaction** groups several operations into one logical unit.

The key idea is:

> **Either all required operations succeed, or the transaction is rolled back.**

A bank transfer is a classic example:

1. subtract money from account A;
2. add money to account B.

Both must succeed together.

24. Auto-Commit

Many JDBC connections begin with auto-commit enabled.

For a manual transaction:

```
connection.setAutoCommit(false);
```

Then:

```
connection.commit();
```

or:

```
connection.rollback();
```

Example 10. Transaction

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class TransferService {

    public void transfer(
            Connection connection,
            int fromId,
            int toId,
            double amount)
            throws SQLException {

        String debitSql =
            "UPDATE account " +
            "SET balance = balance - ? " +
            "WHERE id = ? " +
            "AND balance >= ?";

        String creditSql =
            "UPDATE account " +
            "SET balance = balance + ? " +
            "WHERE id = ?";

        boolean originalAutoCommit =
            connection.getAutoCommit();
```

Example 10 (continued)

```
27
28          try {
29
30              connection.setAutoCommit(
31                  false
32              );
33
34              try (PreparedStatement debit =
35                      connection
36                          .prepareStatement(
37                              debitSql);
38
39                   PreparedStatement credit =
40                      connection
41                          .prepareStatement(
42                              creditSql)) {
43
44                  debit.setDouble(
45                      1,
46                      amount
47                  );
48
49                  debit.setInt(
50                      2,
51                      fromId
52                  );
53
54                  debit.setDouble(
55                      3,
56                      amount
57                  );
58
59                  int debitRows =
60                      debit.executeUpdate();
61
62                  if (debitRows != 1) {
63
64                      throw new SQLException(
65                          "Debit failed."
66                      );
67                  }
68
69                  credit.setDouble(
70                      1,
71                      amount
```

Example 10 (continued)

```
                );

                credit.setInt(
                    2,
                    toId
                );

                int creditRows =
                    credit.executeUpdate();

                if (creditRows != 1) {

                    throw new SQLException(
                        "Credit failed."
                    );
                }

                connection.commit();
            }

        } catch (SQLException e) {

            connection.rollback();

            throw e;

        } finally {

            connection.setAutoCommit(
                originalAutoCommit
            );
        }
    }
}
```

25. ACID Concepts

Transactions are often discussed using ACID:

- **Atomicity**: all-or-nothing execution;
- **Consistency**: constraints remain valid;
- **Isolation**: concurrent transactions are controlled;

- **Durability**: committed changes persist.

26. Batch Processing

When many similar SQL operations must be executed, batch processing can reduce round trips to the database.

Methods include:

```
addBatch()
executeBatch()
clearBatch()
```

Example 11. Batch Insert

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.List;

public class StudentBatchService {

    public void insertAll(
            Connection connection,
            List<Student> students)
            throws SQLException {

        String sql =
            "INSERT INTO student(" +
            "name, age, gpa) " +
            "VALUES (?, ?, ?)";

        try (PreparedStatement statement =
                connection
                    .prepareStatement(sql)) {

            for (Student student :
                    students) {

                statement.setString(
                    1,
                    student.getName()
                );
```

Example 11 (continued)

```
29
30                  statement.setInt(
31                      2,
32                      student.getAge()
33                  );
34
35                  statement.setDouble(
36                      3,
37                      student.getGpa()
38                  );
39
40                  statement.addBatch();
41              }
42
43              int[] counts =
44                  statement.executeBatch();
45
46              System.out.println(
47                  "Batch commands: " +
48                  counts.length
49              );
50          }
51      }
52  }
```

27. The DAO Pattern

DAO stands for **Data Access Object**.

A DAO isolates persistence logic from business logic.

Illustration

```
UI
 |
Service
 |
DAO / Repository
 |
JDBC
 |
Database
```

This separation improves maintainability, testability, modularity, and replaceability of persistence implementations.

28. DAO Interface

A DAO may begin with an interface:

```java
import java.sql.SQLException;
import java.util.List;
import java.util.Optional;

public interface StudentDao {

    int insert(Student student)
            throws SQLException;

    Optional<Student> findById(
            int id)
            throws SQLException;

    List<Student> findAll()
            throws SQLException;

    boolean update(Student student)
            throws SQLException;
```

```
    boolean deleteById(int id)
            throws SQLException;
}
```

Example 12. JDBC DAO Implementation

```
import java.sql.*;
import java.util.Optional;

public class JdbcStudentDao
        implements StudentDao {

    private final Connection connection;

    public JdbcStudentDao(
            Connection connection) {

        this.connection = connection;
    }

    @Override
    public Optional<Student> findById(
            int id)
            throws SQLException {

        String sql =
            "SELECT id, name, age, gpa " +
            "FROM student WHERE id = ?";

        try (PreparedStatement statement =
                connection
                    .prepareStatement(sql)) {

            statement.setInt(
                1,
                id
            );

            try (ResultSet result =
                    statement
                        .executeQuery()) {

                if (result.next()) {

```

Example 12 (continued)

```
                    return Optional.of(
                        map(result)
                    );
                }

                return Optional.empty();
            }
        }
    }

    private Student map(
            ResultSet result)
            throws SQLException {

        return new Student(
            result.getInt("id"),
            result.getString("name"),
            result.getInt("age"),
            result.getDouble("gpa")
        );
    }

    // Other CRUD methods would
    // be implemented similarly.
}
```

29. Repository vs DAO

The terms DAO and repository are sometimes used differently in different architectures.

A simple educational distinction is:

- DAO focuses on persistence operations and SQL-facing concerns;
- repository presents collection-like access to domain objects.

In small applications, the names are sometimes used interchangeably.

The main principle is separation of persistence concerns.

30. Service Layer

Business rules should usually not be placed directly inside GUI classes or SQL code.

```
public class StudentService {

    private final StudentDao dao;

    public StudentService(
            StudentDao dao) {

        this.dao = dao;
    }

    public void register(
            Student student)
            throws SQLException {

        if (student.getGpa() < 0 ||
            student.getGpa() > 4) {

            throw new
                IllegalArgumentException(
                    "Invalid GPA"
                );
        }

        dao.insert(student);
    }
}
```

The service coordinates business rules.

The DAO coordinates persistence.

31. Connection Factory

Connection creation can also be centralized.

Example 13. Connection Factory

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public final class ConnectionFactory {

    private static final String URL =
        "jdbc:postgresql://localhost:5432/university";

    private static final String USER =
        "postgres";

    private static final String PASSWORD =
        "secret";

    private ConnectionFactory() {
    }

    public static Connection open()
            throws SQLException {

        return DriverManager
            .getConnection(
                URL,
                USER,
                PASSWORD
            );
    }
}
```

In real applications, credentials should not be hard-coded into source code.

32. Database Configuration

A simple properties file may contain:

Illustration

```
db.url=jdbc:postgresql://localhost:5432/university
db.user=postgres
db.password=secret
```

Java can read these values using the `Properties` class studied earlier.

Sensitive credentials require careful protection.

33. Connection Pooling

Opening a physical database connection repeatedly can be expensive.

A **connection pool** maintains reusable database connections.

Illustration

```
Application
   |
Connection Pool
 |   |   |
DB  DB  DB
connections
```

The application borrows a connection and returns it to the pool.

34. `DataSource`

The JDBC API defines:

Illustration

```
javax.sql.DataSource
```

It is an abstraction for obtaining connections.

In larger applications, a `DataSource`-based design is commonly preferred over calling `DriverManager` throughout the application.

35. Database Metadata

JDBC can inspect database and result metadata.

Important interfaces include:

- `DatabaseMetaData`
- `ResultSetMetaData`

Example 14. Database Metadata

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;

public class Main {

    public static void main(String[] args)
            throws Exception {

        try (Connection connection =
                DriverManager.getConnection(
                    "jdbc:postgresql://localhost:5432/university",
                    "postgres",
                    "secret")) {

            DatabaseMetaData metadata =
                    connection
                        .getMetaData();

            System.out.println(
                metadata
                    .getDatabaseProductName()
            );

            System.out.println(
                metadata
                    .getDatabaseProductVersion()
            );

            System.out.println(
                metadata
                    .getDriverName()
            );
        }
    }
```

Example 14 (continued)

```
36 }
```

36. Handling SQL `NULL`

SQL `NULL` represents a missing or unknown database value.

JDBC provides methods such as:

```
result.wasNull()
```

and:

```
statement.setNull(
    index,
    Types.INTEGER
);
```

Java `null` and SQL `NULL` belong to different type systems.

37. Dates and Times

Modern Java applications normally use classes from:

Illustration

```
java.time
```

Examples include:

- `LocalDate`
- `LocalTime`
- `LocalDateTime`

Modern JDBC drivers can often map these types using `setObject()` and `getObject()`.

38. Exception Handling with JDBC

JDBC operations commonly throw:

Illustration

`SQLException`

Useful information includes:

- error message;
- SQL state;
- vendor error code;
- chained SQL exceptions.

Example 15. Inspecting `SQLException`

```
catch (SQLException e) {

    System.out.println(
        "Message: " +
        e.getMessage()
    );

    System.out.println(
        "SQLState: " +
        e.getSQLState()
    );

    System.out.println(
        "Error code: " +
        e.getErrorCode()
    );
}
```

Raw database error details should not normally be shown directly to end users.

39. Integrated Application Architecture

A well-organized desktop database application may use:

Illustration

```
app/
  Main.java

model/
  Student.java
  Course.java

dao/
  StudentDao.java
  JdbcStudentDao.java
  CourseDao.java
  JdbcCourseDao.java

service/
  StudentService.java
  CourseService.java

db/
  ConnectionFactory.java

ui/
  StudentFrame.java
  CourseFrame.java
```

This separates user interface, business logic, persistence, database configuration, and domain model.

Example 16. Domain Model

```java
public class Student {

    private int id;
    private String name;
    private int age;
    private double gpa;

    public Student(
            int id,
            String name,
            int age,
```

Example 16 (continued)

```
            double gpa) {

        this.id = id;
        this.name = name;
        this.age = age;
        this.gpa = gpa;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public double getGpa() {
        return gpa;
    }
}
```

The domain model does not contain SQL.

Example 17. Search Method in a DAO

```
public List<Student> searchByName(
        String text)
        throws SQLException {

    String sql =
        "SELECT id, name, age, gpa " +
        "FROM student " +
        "WHERE LOWER(name) LIKE LOWER(?) " +
        "ORDER BY name";

    List<Student> students =
            new ArrayList<>();

    try (PreparedStatement statement =
            connection
                .prepareStatement(sql)) {

```

Example 17 (continued)

```
        statement.setString(
            1,
            "%" + text + "%"
        );

        try (ResultSet result =
                statement
                    .executeQuery()) {

            while (result.next()) {

                students.add(
                    map(result)
                );
            }
        }

    return students;
}
```

40. JDBC in a Swing Application

A Swing interface can call a service, which calls a DAO.

Illustration

```
Button Click
   |
StudentFrame
   |
StudentService
   |
StudentDao
   |
JDBC
   |
Database
```

The GUI should not contain large SQL strings.

Example 18. GUI Event Delegating to a Service

```
saveButton.addActionListener(
    event -> {

        try {

            Student student =
                readStudentFromForm();

            studentService
                .register(student);

            JOptionPane
                .showMessageDialog(
                    frame,
                    "Student saved."
                );

            refreshTable();

        } catch (
                IllegalArgumentException e) {

            JOptionPane
                .showMessageDialog(
                    frame,
                    e.getMessage()
                );

        } catch (SQLException e) {

            JOptionPane
                .showMessageDialog(
                    frame,
                    "Database operation failed."
                );
        }
    }
);
```

41. Avoiding Long Database Work on the EDT

Database calls may block.

In a Swing application, long database work should not freeze the Event Dispatch Thread.

Possible approaches include:

- `SwingWorker`;
- an executor;
- another appropriate background-task design.

After the work finishes, Swing components should be updated safely on the EDT.

42. Testing Database Code

Database code should be tested at several levels:

- test SQL independently;
- test DAO methods;
- test transaction behavior;
- test invalid input;
- test missing records;
- test duplicate constraints;
- test rollback behavior;
- test integration with services and UI.

Use dedicated test data rather than production data.

43. Common Mistakes

43.1 Mistake 1: Concatenating User Input into SQL

Use `PreparedStatement` parameters.

43.2 Mistake 2: Forgetting to Close JDBC Resources

Use try-with-resources.

43.3 Mistake 3: Putting SQL Directly in GUI Event Handlers

Move persistence logic into DAO or repository classes.

43.4 Mistake 4: Hard-Coding Credentials

Use external configuration and appropriate secret management.

43.5 Mistake 5: Forgetting Transactions for Multi-Step Operations

Use explicit transactions where operations must succeed or fail together.

43.6 Mistake 6: Ignoring Rollback

If a transaction fails, rollback should be handled correctly.

43.7 Mistake 7: Treating Database Connections as Cheap Temporary Objects

Use connections carefully; larger applications normally use pooling.

43.8 Mistake 8: Assuming Every Database Uses Identical SQL

SQL dialects and identity-generation syntax can differ.

44. Design Guidelines

When building database-driven Java applications:

1. use `PreparedStatement` for parameterized SQL;
2. use try-with-resources;
3. separate database logic from UI logic;
4. keep business rules in service/domain layers;
5. use transactions for multi-step operations;
6. handle rollback carefully;
7. validate user input before persistence;
8. do not expose database credentials in source code;
9. avoid showing raw SQL errors to end users;
10. use connection pooling in larger applications;
11. use meaningful constraints in the database;
12. test DAO and transaction behavior independently.

45. Summary

In this lecture, we learned that:

- JDBC is the standard Java API for relational database access;
- JDBC drivers connect standard Java calls to a specific database;
- `Connection` represents a database session;
- `PreparedStatement` safely parameterizes SQL;
- `ResultSet` provides query results;
- CRUD corresponds to insert, select, update, and delete operations;
- try-with-resources should manage JDBC resources;
- transactions use commit and rollback;
- generated keys can be returned after insert;
- batch operations can efficiently submit repeated SQL commands;
- DAO/repository patterns separate persistence from business logic;
- connection pooling reuses physical database connections;
- database-driven applications benefit from layered architecture.

46. Review Questions

1. What is a relational database?
2. What is SQL?
3. What does CRUD mean?
4. What is JDBC?
5. What package contains the main JDBC interfaces?
6. What is a JDBC driver?
7. What is a JDBC URL?
8. What is `DriverManager`?
9. What is `Connection`?
10. What is `Statement`?
11. What is `PreparedStatement`?
12. Why is `PreparedStatement` preferred for user-supplied values?
13. What is SQL injection?
14. What is `ResultSet`?
15. What does `ResultSet.next()` do?
16. How are result columns accessed?
17. What JDBC method is commonly used for `SELECT`?
18. What JDBC method is commonly used for `INSERT`, `UPDATE`, and `DELETE`?
19. Why are JDBC parameter indexes one-based?
20. What is a transaction?
21. What is auto-commit?
22. What does `commit()` do?
23. What does `rollback()` do?
24. What does ACID stand for?
25. What is batch processing?
26. How are generated keys retrieved?
27. What is a DAO?
28. What is a repository?
29. What is a service layer?
30. What is connection pooling?
31. What is `DataSource`?
32. What is `DatabaseMetaData`?
33. What is `SQLException`?
34. Why should credentials not be hard-coded?
35. Why should long JDBC operations not block the Swing EDT?

36. Why should SQL be kept out of GUI classes?

47. Programming Exercises

Beginner Level

1. Open a JDBC connection and print whether it is open.
2. Print database product and driver information.
3. Create a table with JDBC.
4. Insert one student using `PreparedStatement`.
5. Select and display all students.
6. Search a student by ID.
7. Update a student's GPA.
8. Delete a student by ID.
9. Count the number of rows in a table.
10. Demonstrate try-with-resources with JDBC objects.

Intermediate Level

1. Build a complete CRUD repository for students.
2. Return `Optional<Student>` from a search method.
3. Search students by partial name.
4. Sort students using SQL `ORDER BY`.
5. Insert a student and return the generated ID.
6. Insert several students with batch processing.
7. Create a transaction involving two SQL updates.
8. Demonstrate rollback when one transaction step fails.
9. Read database configuration from a properties file.
10. Create a reusable connection factory.

Advanced Level

1. Build DAO interfaces for students and courses.
2. Implement JDBC DAOs for both entities.
3. Create a service layer above the DAOs.
4. Implement student-course registration using a many-to-many relation.
5. Use a transaction when registering a student and updating course capacity.
6. Build a Swing CRUD interface connected to JDBC.
7. Add search and filtering to the GUI.
8. Perform long database queries outside the Swing EDT.
9. Replace direct connection creation with a `DataSource`-based design.
10. Develop a complete layered University Management application.

48. Final Course Project

Project: Database-Driven University Student Management System

Develop an integrated Java application that combines the major topics studied throughout the course.

Suggested package structure:

Illustration

```
app/
  Main.java

model/
  Student.java
  Course.java
  Registration.java

dao/
  StudentDao.java
  CourseDao.java
  RegistrationDao.java
  JdbcStudentDao.java
  JdbcCourseDao.java
  JdbcRegistrationDao.java

service/
  StudentService.java
  CourseService.java
```

```
  RegistrationService.java

db/
  ConnectionFactory.java

ui/
  MainFrame.java
  StudentPanel.java
  CoursePanel.java
  RegistrationPanel.java
```

Functional requirements:

1. Manage students:
   - add;
   - view;
   - update;
   - delete;
   - search.
2. Manage courses:
   - add;
   - view;
   - update;
   - delete.
3. Register students for courses.
4. Prevent duplicate registrations.
5. Enforce course capacity.
6. Use transactions for multi-step registration logic.
7. Display data in Swing tables.
8. Validate all form input.
9. Use `PreparedStatement`.
10. Use exception handling with user-friendly messages.
11. Read database configuration from a properties file.
12. Keep SQL outside GUI classes.
13. Use interfaces for DAO contracts.
14. Use collections where temporary in-memory processing is needed.
15. Perform blocking database work away from the Swing EDT when operations may take noticeable time.

Technical requirements:

- object-oriented design;
- encapsulation;
- inheritance or interfaces where meaningful;
- exception handling;
- collections and generics;
- file/properties configuration;
- Swing GUI;
- JDBC persistence;
- transactions;
- modular package structure.

Submission requirements:

- all Java source files;
- SQL schema creation script;
- sample database records;
- configuration file;
- screenshots of the GUI;
- short architecture diagram;
- explanation of DAO and service layers;
- explanation of transaction handling;
- explanation of SQL injection prevention;
- short final report describing the application.

49. Key Terms

Database
Organized persistent data storage system.

Relational database
Database organized around tables and relationships.

SQL
Structured Query Language.

JDBC
Java Database Connectivity API.

JDBC driver
Database-specific implementation enabling JDBC communication.

`Connection`
JDBC interface representing a database session.

`Statement`
JDBC interface for executing SQL text.

`PreparedStatement`
Parameterized SQL statement abstraction.

`ResultSet`
Cursor-based representation of query results.

CRUD
Create, Read, Update, Delete.

Transaction
Group of database operations treated as one logical unit.

Commit
Permanently accepts transaction changes.

Rollback
Cancels uncommitted transaction changes.

Batch
Group of SQL commands submitted together.

DAO
Data Access Object.

Repository
Persistence abstraction providing domain-oriented data access.

Connection pool
Reusable collection of database connections.

`DataSource`
Standard JDBC abstraction for obtaining connections.

SQL injection
Security vulnerability caused by unsafe construction of SQL from untrusted input.