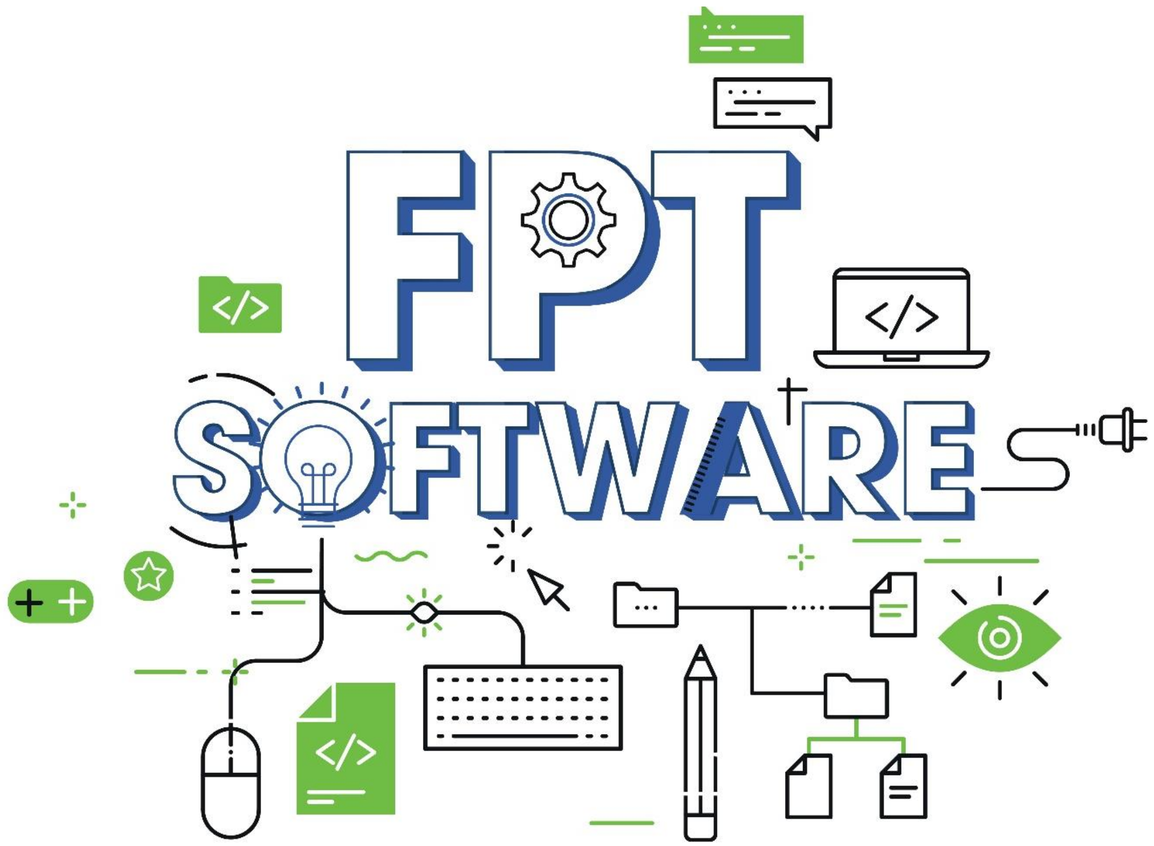
# Database Programing with JDBC

*Fsoft Academy*



## Lesson Objectives

- ♦ **Understand** the fundamentals of JDBC and its role in Java applications.
- ♦ **Be able to connect** to a database using JDBC.
- ♦ **Execute** SQL statements using JDBC (INSERT, UPDATE, DELETE, SELECT).
- ♦ **Process** the results of SQL statements using ResultSet
- ♦ **Handle** errors and exceptions.
- ♦ **Be able use** advanced JDBC features such as: using prepared statements, callable statements, transactions

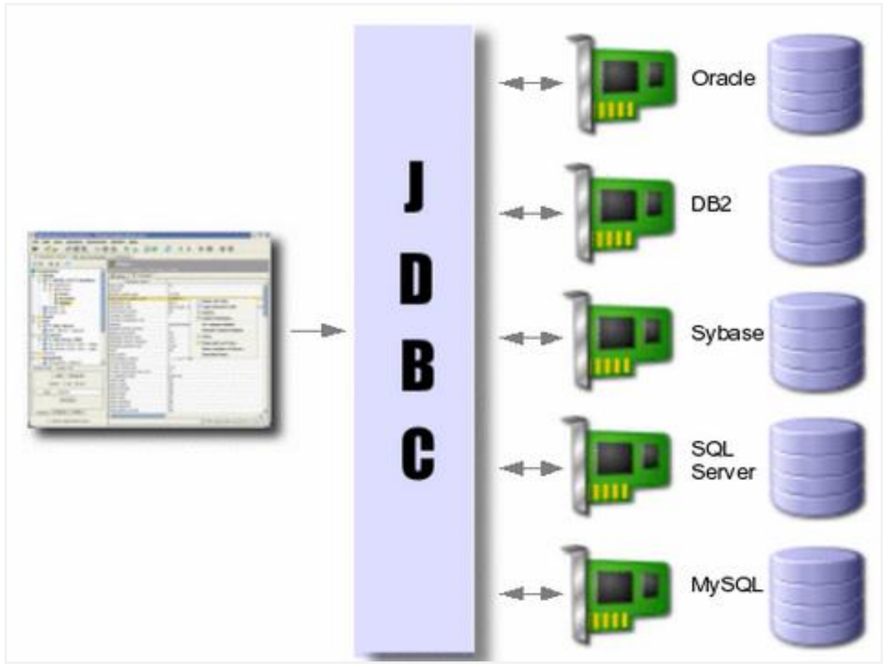
## Table of contents

Section 1

## Java JDBC Tutorial

## Overview

- **JDBC API** (Java Database Connectivity) allows connecting Java to databases
- Database access is the same for all database vendors
- **JVM** (Java virtual Machine) uses **JDBC driver** to translate JDBC calls to vendor specific database  calls.

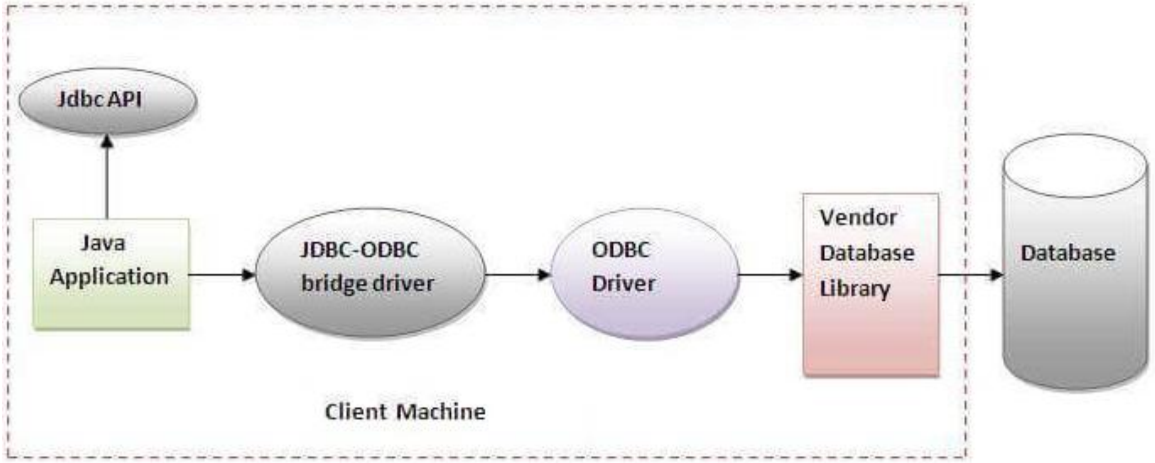


## Overview

- **JDBC uses drivers to connect to databases.**
- **Four JDBC driver types:**
  - ✓ JDBC-ODBC Bridge
  - ✓ Native Driver
  - ✓ Network Protocol Driver
  - ✓ Thin Driver
- A JDBC driver is Java classes implementing JDBC interfaces for a specific database.

## JDBC-ODBC bridge driver

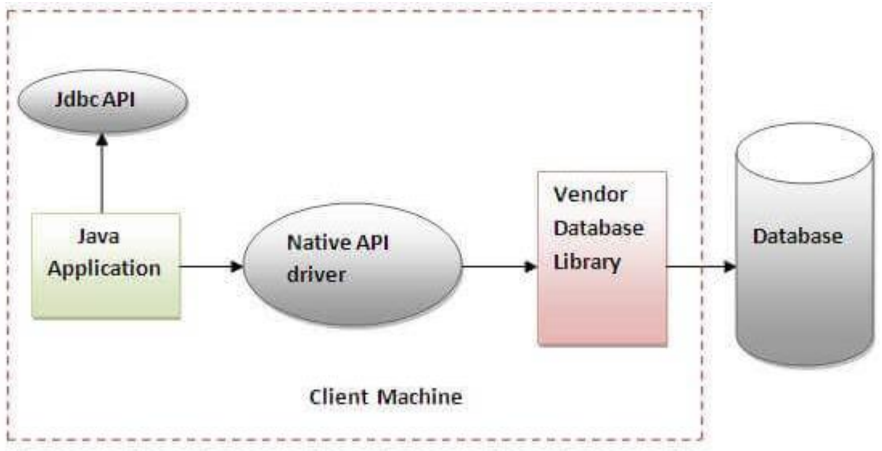
- **JDBC-ODBC bridge driver** uses ODBC driver (Open Database Connectivity), converts JDBC to ODBC.
- Discouraged due to thin driver.



- *Oracle does not support JDBC-ODBC Bridge from Java 8, recommends using vendor JDBC drivers instead.*

## Native-API driver

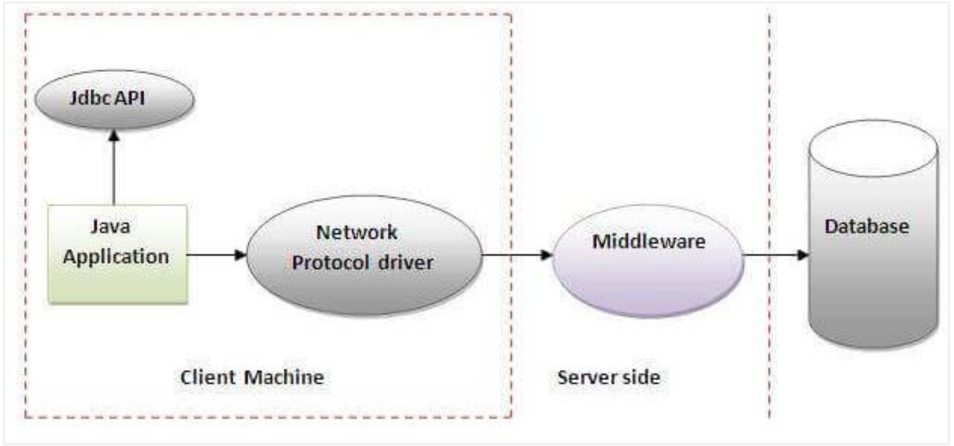
- **Native API driver** uses database client libraries, converts JDBC to native calls.
  - ✓Not entirely Java.



- **Advantages**:
  - ✓Better performance than JDBC-ODBC bridge.
- **Disadvantages**:
  - ✓Needs installing on each client.
  - ✓Requires database vendor client library on client.

# Network Protocol driver

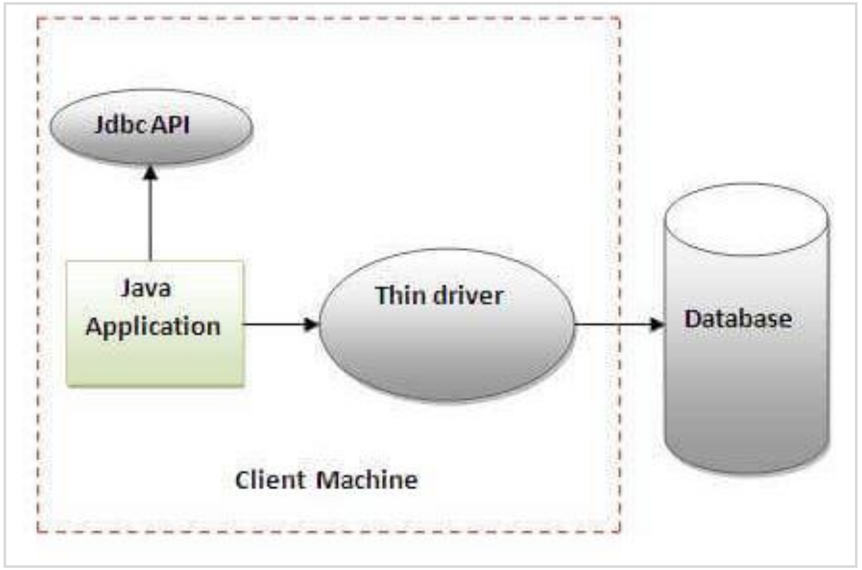
- **Network Protocol driver** uses middleware to convert JDBC to vendor protocol. Fully written in Java.



- **Advantages**:
  - ✓ No client library needed since
  - ✓ app server can handle auditing, load balancing, logging, etc.
- **Disadvantages**:
  - ✓ Requires network support on client.
  - ✓ Database-specific coding in middleware.
  - ✓ Maintenance more costly due to database-specific middleware coding.

## Thin driver

- **Thin driver** converts JDBC directly to vendor protocol. Fully written in Java.



- **Advantages:**
  - ✓Better performance than other drivers.
  - ✓No software required on client or server.
- **Disadvantages:**
  - ✓Drivers dependent on database.

# Before you start…

- To connect to a MySQL database using Java, you should use the "Thin Driver" or "Type-4 Driver." The Thin Driver for MySQL is provided by MySQL Connector/J, which is the official JDBC library for MySQL.
- Here are the steps to get started with MySQL Connector/J:
  - ✓ **Download MySQL Connector/J**: download MySQL Connector/J from the official MySQL website or through a build tool like Maven or Gradle.
  - ✓ **Add MySQL Connector/J to Your Project:** After downloading, you need to add the MySQL Connector/J JAR file to your Java project.
  - ✓ **Connect to the Database**: Use your connection information (URL, username, password) to establish a connection to the MySQL database.

## Before you start…

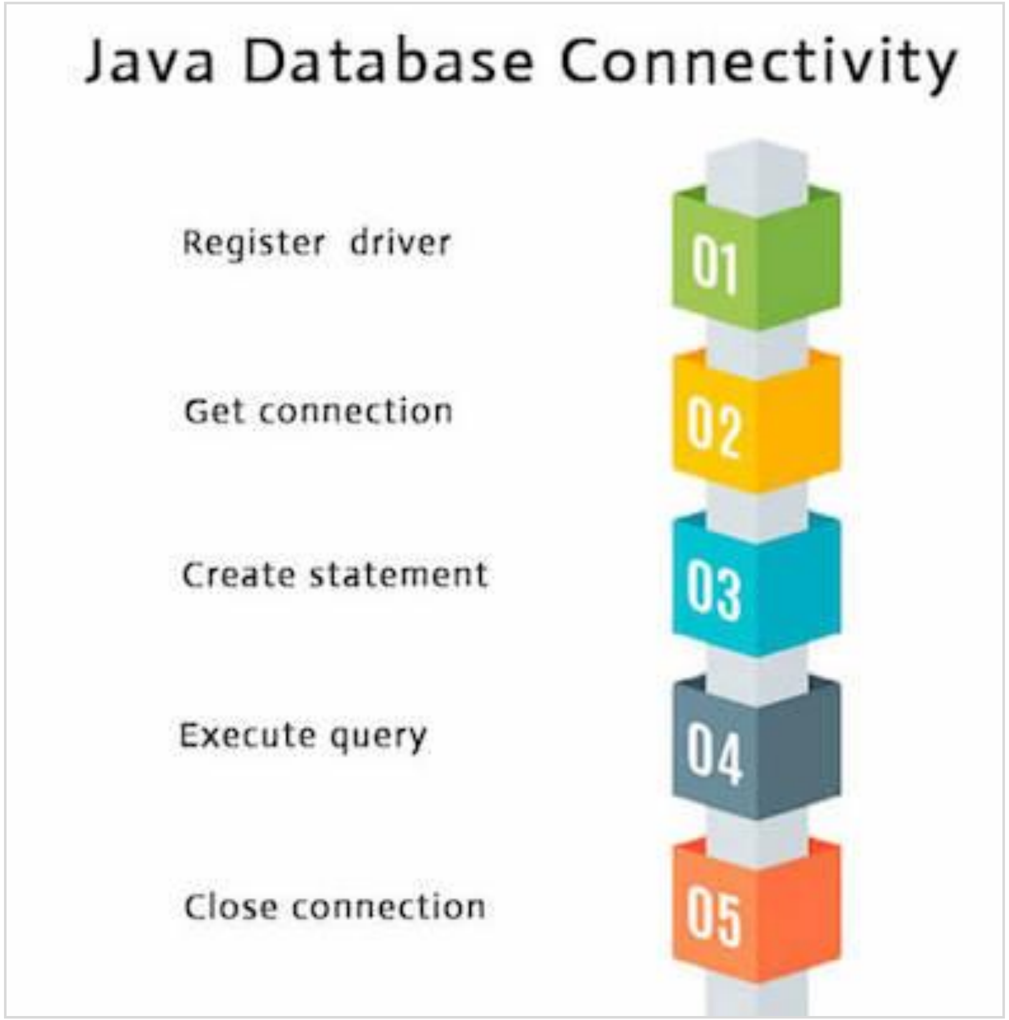
- If you have Maven project, just need to update **pom.xml**

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.27</version> <!-- Specify the version you want to use -->
</dependency>
```

Section 2

## Working steps

## Working steps



## 1. Register the driver class

- **Class.forName()** registers the driver class, dynamically loads it
- Registering Oracle Driver:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- Registering SQLServer Driver:

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

- Registering MySQL Server Driver:

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

- *Note: Since JDBC 4.0, registering driver is optional - just add vendor jar to classpath and it loads automatically.*

## 2. Create connection

- **DriverManager.getConnection()** establishes database connection.
- **Syntax:**

```
public static Connection getConnection(String url) throws SQLException
public static Connection getConnection(String url, String user, String password)
```

- Examples:
  - ✓Oracle:

```
String url = "jdbc:oracle:thin:@localhost:1521:xe";
Connection con = DriverManager.getConnection(url, "system", "password");
```

  - ✓MySQL:

```
String url = "jdbc:mysql://localhost:3306/ebookshop";
Connection conn = DriverManager.getConnection(url, "myuser", "xxxx");
```

  - ✓SQL Server:

```
String url = "jdbc:sqlserver://localhost:1433;databaseName=Fsoft_Training";
Connection conn = DriverManager.getConnection(url, "system", "password");
```

# 3. Create Access Statement

- **Connection.createStatement()** creates a Statement to execute queries.
  - ✓ Used for general database access.
  - ✓ Useful for static SQL at runtime.
  - ✓ The Statement interface cannot accept parameters.
- **Syntax:**

```
Statement stmt = null;
try {
      stmt = conn.createStatement(); // or
      stmt = con.createStatement(ResultSetType, ConcurencyType);
  } catch (SQLException e) {
  } finally {
      if (stmt != null) {
           stmt.close();
      }
}
```

## 4. Execute the query

- **Statement.executeQuery()** executes a query, returns a ResultSet of records.
- **Syntax**:

```
public ResultSet executeQuery(String sql) throws SQLException
```

- **Example:**

```
ResultSet rs = stmt.executeQuery("SELECT * FROM Employee");

while(rs.next()) {
    System.out.println(rs.getInt(1) + " " + rs.getString(2));
}
```

## 5. Close the connection object

- Closing the **Connection** also closes the **Statement** and **ResultSet**.
- **Connection.close()** closes the database connection.
- **Syntax:**

```
public void close() throws SQLException
```

- **Example**:

```
con.close();
```

# Try-with-resource

- Since Java 7, JDBC has ability to use **try-with-resources** statement to automatically close resources of type Connection, ResultSet, and Statement.

```
// Use try-with-resources to automatically close resources
try (Connection connection = DriverManager.getConnection(jdbcUrl, username, password);
     Statement statement = connection.createStatement();
     ResultSet resultSet = statement.executeQuery("SELECT * FROM users");) {

    while (resultSet.next()) {
        int id = resultSet.getInt("id");
        String name = resultSet.getString("name");
        String email = resultSet.getString("email");
        System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

Section 3

## DriverManager class

# DriverManager class

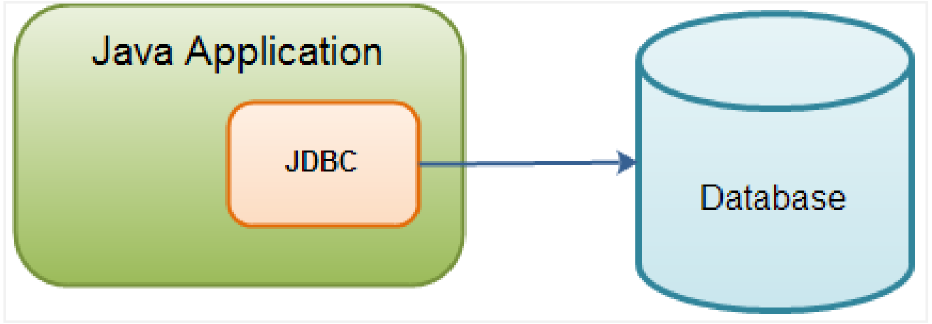
- The **DriverManager** class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver.
- The **DriverManager** maintains registered Driver classes via `DriverManager.registerDriver().`
- **Methods**:
  - ✓ **registerDriver**(Driver) registers a driver with DriverManager.
  - ✓ **deregisterDriver**(Driver) deregisters a driver from DriverManager.
  - ✓ **getConnection**(String) establishes a connection for a given URL.
  - ✓ **getConnection**(String, String, String) establishes a connection for a given URL, username, and password.

# Example

```java
public class RegisterExample {
    public static void main(String[] args) {
        try {

            // Đăng ký Driver MySQL
            Driver mysqlDriver = new com.mysql.cj.jdbc.Driver();
            DriverManager.registerDriver(mysqlDriver);

            // Thiết lập thông tin kết nối cơ sở dữ liệu
            String url = "jdbc:mysql://localhost:3306/fsoft_db";
            String username = "root";
            String password = "1234567890";

            // Kết nối đến cơ sở dữ liệu
            Connection connection = DriverManager.getConnection(url, username, password);

            // Thực hiện các thao tác với cơ sở dữ liệu
            // ...

            // Đóng kết nối
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

## Connection Interface

- A Connection is the session between a Java app and a database.
- All SQL statements are executed and results are returned with in the context of a Connection object.
- Connections default to committing after executing queries.
- The Connection interface factories **Statements**, **PreparedStatements**, and **DatabaseMetaData**.
- You can also use it to retrieve the metadata of a database like name of the database product, name of the JDBC driver, major and minor version of the database etc.



# Connection Interface

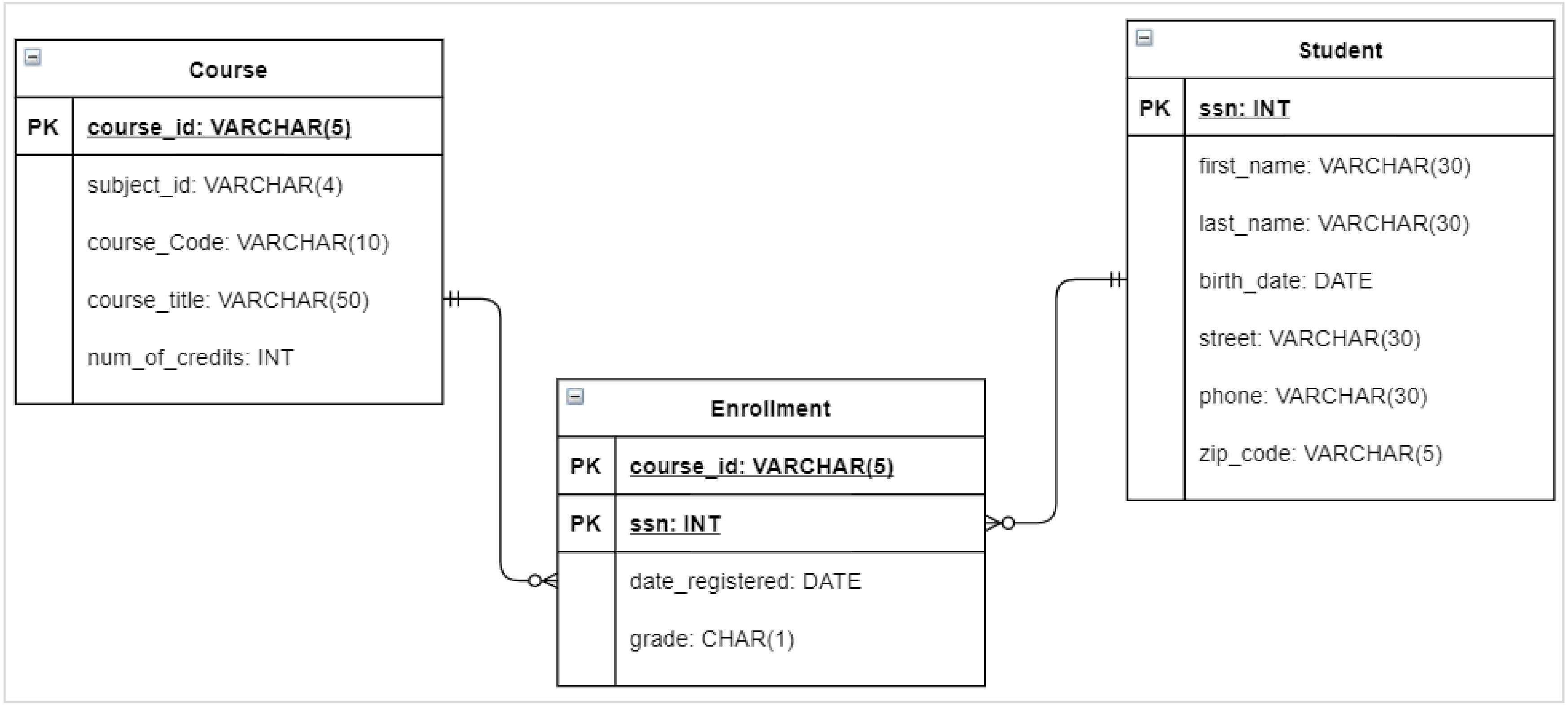
| Methods | Descriptions |
|---|---|
| Statement **createStatement**() throws SQLException | This method creates a **java.sql.Statement** object which can be used to execute SQL queries. |
| PreparedStatement **prepareStatement**(String sql) throws SQLException | This method creates a **java.sql.PreparedStatement** object which can be used to execute the parameterized SQL statements. |
| CallableStatement **prepareCall**(String sql) throws SQLException | This method creates **java.sql.CallableStatement** object which can be used to call stored procedures of the database. |
| void **setAutoCommit**(boolean autoCommit) throws SQLException | This method sets the **auto-commit** mode of this Connection object.<br>✓ If the auto-commit mode of a Connection object is true, then all SQL statements will be executed and committed as individual transactions.<br>✓ If the auto-commit mode is false then all SQL statements will be grouped in transactions.<br>✓ By default, auto-commit mode of a Connection object is true. |

# Connection Interface

| Methods | Descriptions |
|---|---|
| void **commit**() throws SQLException | This method makes all previous changes made to database since last **commit**() OR **rollback**() as permanent. This method should be used only when auto-commit mode of Connection object is false. |
| void **rollback**() throws SQLException | This method erases all changes made to database in the current transaction. This method also should be called when auto-commit mode of a Conncetion object is false. |
| boolean **isValid**(int timeout) throws SQLException | This method checks whether the current Connection object is still valid or is it closed. |
| boolean **isClosed**() throws SQLException | This method checks whether the current Conncetion object is closed or not. |
| void **close**() throws SQLException | This method closes the current Conncetion object and releases the resources held by it. |

## DB Sample

- Create a Database for Training Management System (**TrainingDB**) includes the following tables as:



## DB Sample

```
CREATE TABLE Course (
    course_id VARCHAR(5),
    subject_id VARCHAR(4),
    course_code VARCHAR(10),
    course_title VARCHAR(50),
    num_of_credits INT
);
CREATE TABLE Student (
    ssn INT PRIMARY KEY,
    first_name VARCHAR(30),
    last_name VARCHAR(30),
    birth_date DATE,
    street VARCHAR(30),
    phone VARCHAR(30),
    zip_code VARCHAR(5)
);
```

```
CREATE TABLE Enrollment (
    course_id VARCHAR(5),
    ssn INT,
    date_registered DATE,
    grade CHAR(1),
    PRIMARY KEY (course_id, ssn)
);
```

Section 4

## JDBC Statement

## Statement Interface

- The **Statement** interface provides methods to execute queries.
- Statement is a factory for ResultSet objects. It has factory methods to get ResultSet objects.
- **Create Statement:**

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

Statement statement = connection.createStatement(resultSetType, resultSetConcurrency);

Statement statement = connection.createStatement(resultSetType, esultSetConcurrency,
                                                              resultSetHoldability);
```

# Statement interface

- Statement's **methods**:
  - ✓ **boolean execute(String SQL)** : may be any kind of SQL statement. Returns a boolean value of true if a ResultSet object can be retrieved; false if the first result is an update count or there is no result.
  - ✓ **int executeUpdate(String SQL)** : Returns the numbers of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
  - ✓ **ResultSet executeQuery(String SQL)** : Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.
  - ✓ **public int[] executeBatch():** is used to execute batch of commands.

## Examples

- **Example 1: Execute a SELECT query via a Statement**

```
// Create and execute an SQL statement that returns some data.

String SQL1 = "SELECT TOP 10 * FROM Student";

Statement stmt=conn.createStatement();

//ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE

ResultSet rs = stmt.executeQuery(SQL);
```

- **Example 2: Execute an INSERT via a Statement**

```
String SQL2 = "INSERT INTO Student (ssn, first_name, last_name, street, phone, zip_code)
VALUES (12345, 'John Doe', '120 Hanover Sq', '8765666666', '12209')";
Statement stmt = conn.createStatement();
int no_of_row = stmt.executeUpdate(SQL);
```

# Example: Using Java Try With Resources

```
public class TestResultSet {
   private static final String DB_URL = "jdbc:mysql://localhost:3306/TrainingDB";
   private static final String DB_USER = "root", DB_PASSWORD = "";

   public static void main(String[] args) throws SQLException {
      try (Connection conn = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
          Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);) {
          String query = "SELECT * FROM Student";
          ResultSet resultSet = stmt.executeQuery(query);
          resultSet.last(); // Move to the last row
          System.out.println(
          "Last Row - Name: " + resultSet.getString("first_name") + "\t" +
                                 resultSet.getString("last_name"));
          resultSet.first(); // Move to the first row
          System.out.println(
          "First Row - Name: " + resultSet.getString("first_name") + "\t" +
                                 resultSet.getString("last_name"));
          resultSet.absolute(3); // Move to a specific row
          System.out.println("Row 3 - Name: " + resultSet.getString("first_name") + "\t" +
                                                resultSet.getString("last_name"));
          resultSet.relative(-2); // Move backward by two rows
          System.out.println("Row 1 - Name: " + resultSet.getString("first_name") + "\t" +
                                                resultSet.getString("last_name"));
      }
   }
}
```

Once the try block exits, the **Statement** will be closed automatically.

## Retrieve Data & Close Connection

- **Retrieve data:**

```
// Iterate through the data in the result set and display it.
while (rs.next()) {
    System.out.println(rs.getInt(1) + "\t" + rs.getString(2)+"\t"+rs.getInt(3));
}
```

- **Close connection:**

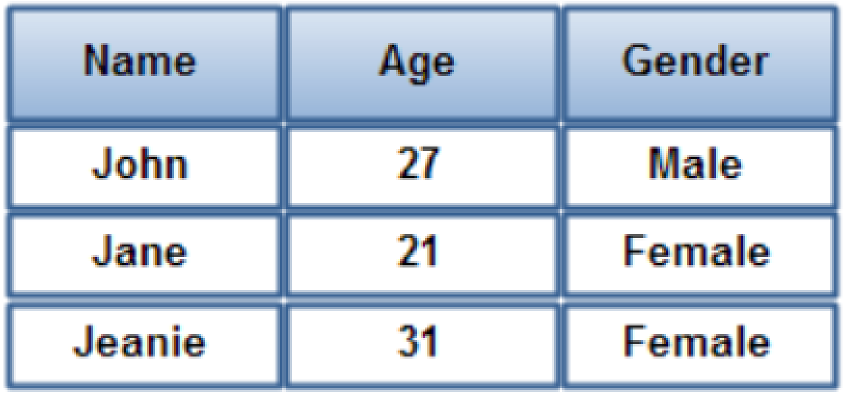
```
statement.close();
conn.close();
```

Section 5

## JDBC ResultSet

## Overview

- The **ResultSet** interface represents the result set of a database query. It contains records made up of columns, where each **record** has the same columns but column values can be null.
- The **ResultSet** is iterated to inspect the query results.



| Name | Age | Gender |
|---|---|---|
| John | 27 | Male |
| Jane | 21 | Female |
| Jeanie | 31 | Female |

**ResultSet example - records with columns**

# Creating a ResultSet

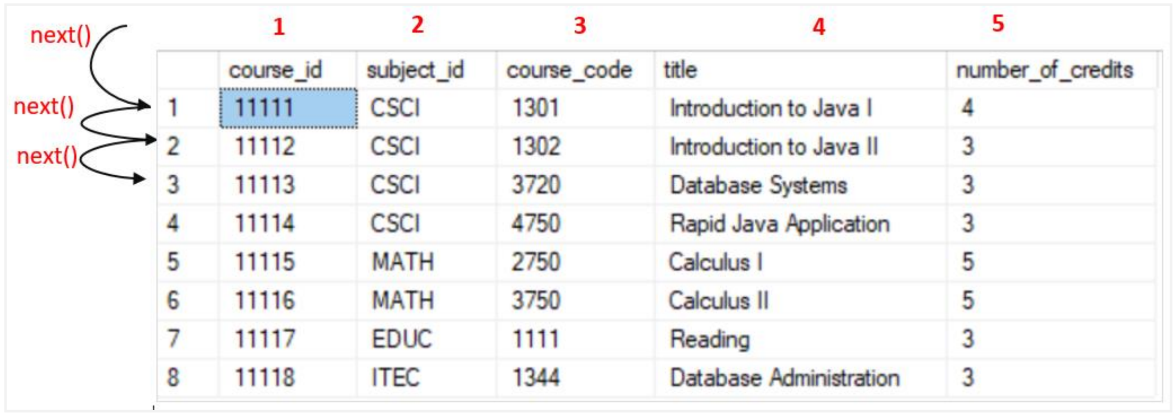
- Create a **ResultSet** by executing a **Statement** or **PreparedStatement**

```
Statement statement = connection.createStatement();
ResultSet result = statement.executeQuery("SELECT * FROM Course");
```

- Or:

```
String selectQuery = "SELECT * FROM Course";
PreparedStatement statement = connection.prepareStatement(selectQuery);
ResultSet result = statement.executeQuery();
```

- ResultSet data:



| | course_id | subject_id | course_code | title | number_of_credits |
|---|---|---|---|---|---|
| 1 | 11111 | CSCI | 1301 | Introduction to Java I | 4 |
| 2 | 11112 | CSCI | 1302 | Introduction to Java II | 3 |
| 3 | 11113 | CSCI | 3720 | Database Systems | 3 |
| 4 | 11114 | CSCI | 4750 | Rapid Java Application | 3 |
| 5 | 11115 | MATH | 2750 | Calculus I | 5 |
| 6 | 11116 | MATH | 3750 | Calculus II | 5 |
| 7 | 11117 | EDUC | 1111 | Reading | 3 |
| 8 | 11118 | ITEC | 1344 | Database Administration | 3 |

```
while (result.next()) {
    System.out.println(result.getString(1)
          + "\t" + result.getString(2)
          + "\t" + result.getString(3)
          + "\t" + result.getString(4)
          + "\t" + result.getInt(5));
}
```

## ResultSet Type, Concurrency, Holdability

- When creating a **ResultSet**, you can set:
  - ✓ Type
  - ✓ Concurrency
  - ✓ Holdability

```
// For use with ResultSet only
// No "previous" method using, no update
Statement stmt = conn.createStatement(
        ResultSet.TYPE_FORWARD_ONLY,
        ResultSet.CONCUR_READ_ONLY,
        ResultSet.HOLD_CURSORS_OVER_COMMIT);

// With "previous" method using, update
Statement statement = conn.createStatement(
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
```

# JDBC ResultSet

- **ResultSet Types:**
  - ✓ **TYPE_FORWARD_ONLY** (default) - cursor moves forward only
  - ✓ **TYPE_SCROLL_INSENSITIVE** - cursor scrolls both ways, not sensitive to database changes
  - ✓ **TYPE_SCROLL_SENSITIVE** - cursor scrolls both ways, sensitive to database changes
- **ResultSet Concurrency:**
  - ✓ **ResultSet.CONCUR_READ_ONLY** (default) - Creates a read-only ResultSet
  - ✓ **ResultSet.CONCUR_UPDATABLE** - Creates an updatable ResultSet
- **ResultSet Holdability:**
  - ✓ **ResultSet.HOLD_CURSORS_OVER_COMMIT** - ResultSet remains open after transaction commit. Shows changes by others.
  - ✓ **ResultSet.CLOSE_CURSORS_AT_COMMIT** - ResultSet closes after transaction commit. Doesn't show changes by others.

# ResultSet Methods

- **beforeFirst**() - Moves before first row
- **afterLast**() - Moves after last row
- **first**() - Moves to first row
- **last**() - Moves to last row
- **absolute**(int) - Moves to specified row
- **relative**(int) - Moves cursor rows forward or backward

- **previous**() - Moves to previous row, false if off result set
- **next**() - Moves to next row, false if no more rows
- **getRow**() - Returns current row number
- **moveToInsertRow**() - Moves to special insert row, remembering current location
- **moveToCurrentRow**() - Moves back to current row from insert row

## JDBC ResultSet

- Getting Values from ResultSet:
  - ✓ **getInt**(String) - Gets int from column name
  - ✓ **getInt**(int) - Gets int from column index (starting at 1)
  - ✓ **getXXX**(int) - Gets value from column index based on XXX type

# JDBC ResultSet

- **Updating ResultSet:** Has update methods for each data type like get methods:
  - ✓ By column name
  - ✓ By column index
- **Examples:**
  - ✓ **updateString**(int, String) - Updates String by column index
  - ✓ **updateString**(String, String) - Updates String by column name

# ResultSet Example

```
public class DatabaseConnection {
    private static final String JDBC_URL =
                "jdbc:mysql://localhost:3306/student_management";
    private static final String USERNAME = "root";
    private static final String PASSWORD = "1234567890";

    public static Connection getConnection() {
        Connection connection = null;

        try {
            connection = DriverManager.getConnection(JDBC_URL,
                    USERNAME, PASSWORD);
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return connection;
    }
}
```

```
public class CourseDAO {
    public List<Course> findCourseByName(String name) {
        List<Course> courses = new ArrayList<>();
        String sql = "SELECT * FROM Course WHERE course_title LIKE ?";

        try (Connection connection = DatabaseConnection.getConnection();
             PreparedStatement statement = connection.prepareStatement(sql)) {
            statement.setString(1, "%" + name + "%");
            try (ResultSet resultSet = statement.executeQuery()) {
                while (resultSet.next()) {
                    Course course = new Course();
                    course.setCourseId(resultSet.getString("course_id"));
                    course.setSubjectId(resultSet.getString("subject_id"));
                    course.setCourseCode(resultSet.getString("course_code"));
                    course.setCourseTitle(resultSet.getString("course_title"));
                    course.setNumOfCredits(resultSet.getInt("num_of_credits"));
                    courses.add(course);
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return courses;
    }
    // Other methods
}
```

## ResultSet Example (cont)

```
public class CourseMain {
    public static void main(String[] args) {
        CourseDAO courseDao = new CourseDAO();
        List<Course> courses = courseDao.findCourseByName("Computer");
        if (courses.isEmpty()) {
            System.out.println("No courses found.");
        } else {
            System.out.println("Courses found:");
            for (Course course : courses) {
                System.out.println("Course ID: " + course.getCourseId());
                System.out.println("Subject ID: " + course.getSubjectId());
                System.out.println("Course Code: " + course.getCourseCode());
                System.out.println("Course Title: " + course.getCourseTitle());
                System.out.println("Number of Credits: " + course.getNumOfCredits());
                System.out.println();
            }
        }
    }
}
```

# JDBC Update using ResultSet

```
public class ResultSetUpdate {
    public static void main(String[] args) {
        String courseId = "00001";

        try (Connection connection = DatabaseConnection.getConnection();
             Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
             ResultSet resultSet = statement.executeQuery("SELECT * FROM Course WHERE course_id = '" + courseId + "'")) {

            if (resultSet.next()) {
                // Update the course title and number of credits
                resultSet.updateString("course_title", "Updated Course Title");
                resultSet.updateInt("num_of_credits", 4);
                resultSet.updateRow();
                System.out.println("Course updated successfully.");

                // Delete the row
                // resultSet.deleteRow();  System.out.println("Course deleted successfully.");
            } else {
                System.out.println("Course with ID " + courseId + " not found.");
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Section 6

## JDBC PreparedStatement (with Parameter)

# PreparedStatement Interface

- **PreparedStatement**:
  - ✓ Extends **Statement** for parameterized queries
  - ✓ Improves performance - query compiled once

```
public interface PreparedStatement extends Statement {

}
```

- Created via **Connection.prepareStatement()**

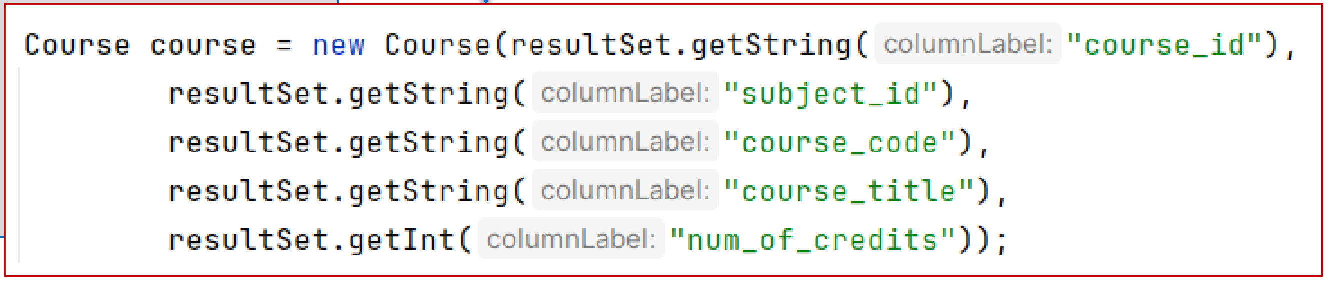
```
public PreparedStatement prepareStatement(String query)  throws SQLException{}
```

- Allows dynamic parameter binding

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
} catch (SQLException e) {//TODO
} finally {//TODO}
```

# Example

```java
public Course getCourseById(String courseId) {
    Course course = null;
    String sql = "SELECT * FROM Course WHERE course_id = ?";

    try (Connection connection = DatabaseConnection.getConnection();
         PreparedStatement statement = connection.prepareStatement(sql)) {

        statement.setString(1, courseId);
        ResultSet resultSet = statement.executeQuery();

        if (resultSet.next()) {

            course = new Course();
            course.setCourseId(resultSet.getString("course_id"));
            course.setSubjectId(resultSet.getString("subject_id"));
            course.setCourseCode(resultSet.getString("course_code"));
            course.setCourseTitle(resultSet.getString("course_title"));
            course.setNumOfCredits(resultSet.getInt("num_of_credits"));
        }

    } catch (SQLException e) {
        e.printStackTrace();
    }

    return course;
}
```



```java
Course course = new Course(resultSet.getString( columnLabel: "course_id"),
        resultSet.getString( columnLabel: "subject_id"),
        resultSet.getString( columnLabel: "course_code"),
        resultSet.getString( columnLabel: "course_title"),
        resultSet.getInt( columnLabel: "num_of_credits"));
```

# PreparedStatement Interface

- **Methods:**
  - ✓ **setInt**(int, int) - Sets integer parameter
  - ✓ **setString**(int, String) - Sets String parameter
  - ✓ **setFloat**(int, float) - Sets float parameter
  - ✓ **setDouble**(int, double) - Sets double parameter
  - ✓ **executeUpdate**() - Executes DML queries
  - ✓ **executeQuery**() - Executes SELECT, returns ResultSet
- **Example:**

```
pstmt.setInt(1,23);
pstmt.setString(2,"Roshan");
pstmt.setString(3,"CEO");
pstmt.executeUpdate();
```

## PreparedStatement Example

- **addCourse**() method:

```
public void addCourse(Course course) {
    String sql = "INSERT INTO Course (course_id, subject_id, course_code, course_title, num_of_credits) " +
            "VALUES (?, ?, ?, ?, ?)";
    try (Connection connection = DatabaseConnection.getConnection();
         PreparedStatement statement = connection.prepareStatement(sql)) {
        statement.setString(1, course.getCourseId());
        statement.setString(2, course.getSubjectId());
        statement.setString(3, course.getCourseCode());
        statement.setString(4, course.getCourseTitle());
        statement.setInt(5, course.getNumOfCredits());
        statement.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Section 7

## JDBC CallableStatement

# CallableStatement Interface

- **CallableStatement**:
  - ✓ Calls stored procedures and functions
  - ✓ Can execute business logic on database by using stored procedures ==> improves performance as precompiled
- **Example:**
  - ✓ You can create a function to get employee age from date of birth. It takes date as input and returns age as output, executing business logic on the database.
  - ✓ Get instance via **Connection.prepareCall()**

```
CallableStatement stmt=con.prepareCall("{call myprocedure(?,?)}");
```

## CallableStatement Example

- In MySQL, create a new User-Stored Procedure:

```
DELIMITER $$
CREATE PROCEDURE usp_UpdateCourse( IN p_course_id VARCHAR(5),
   IN p_subject_id VARCHAR(4),   IN p_course_code VARCHAR(10),
   IN p_title VARCHAR(50),  IN p_number_of_credits INT,
   OUT status VARCHAR(50))
BEGIN
              DECLARE v_rowcount INT;
              SET v_rowcount = 0;
              -- Update the course
              UPDATE Course  SET  subject_id = p_subject_id, course_title = p_title,
              num_of_credits = p_number_of_credits   WHERE course_code = p_course_code;
              -- Check if the update affected any rows
              SELECT ROW_COUNT() INTO v_rowcount;
              IF v_rowcount > 0 THEN
                            SET status = 'Course updated successfully.';
              ELSE
                            SET status = 'Course update failed.';
              END IF;
END $$
DELIMITER ;
```

## CallableStatement Example

```
public String update(Course course) throws SQLException {
    // Initialize CallableStatement and Connection
    try (Connection connection = DatabaseConnection.getConnection();
         CallableStatement callableStatement = connection.prepareCall("{CALL usp_UpdateCourse(?,?,?,?,?,?)}")) {
        // Set input parameters
        callableStatement.setString(1, course.getCourseId());
        callableStatement.setString(2, course.getSubjectId());
        callableStatement.setString(3, course.getCourseCode());
        callableStatement.setString(4, course.getCourseTitle());
        callableStatement.setInt(5, course.getNumOfCredits());
        // Register an output parameter for the stored procedure
        callableStatement.registerOutParameter(6, Types.VARCHAR); // status

        // Execute the stored procedure
        callableStatement.execute();

        // Get the result from the output parameter
        String result = callableStatement.getString(6);
        // Return true if the update was successful
        return result;
    }
}
```

## CallableStatement Example

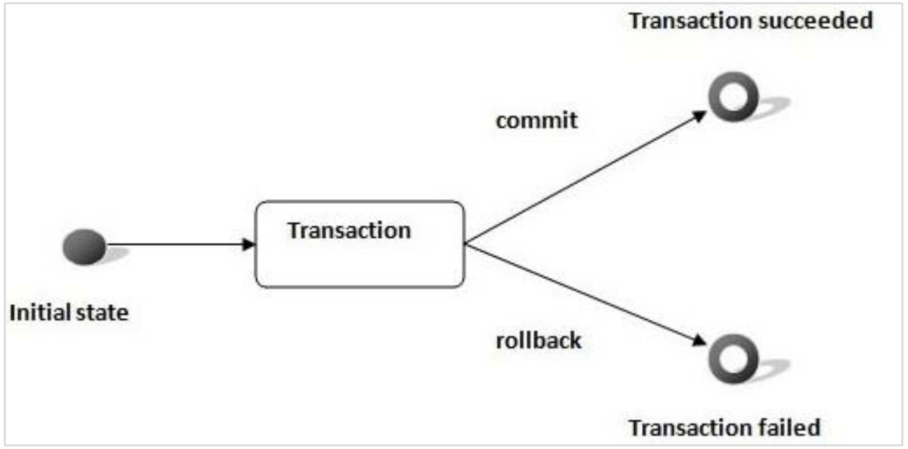
```
public class CourseUpdater {
    public static void main(String[] args) {
        CourseDAO courseDAO = new CourseDAO();

        // Create a Course object with updated information
        Course updatedCourse = new Course();
        updatedCourse.setCourseId("00001");
        updatedCourse.setSubjectId("001");
        updatedCourse.setCourseCode("CS101");
        updatedCourse.setCourseTitle("Updated Course Title");
        updatedCourse.setNumOfCredits(4);

        String updateResult;
        try {
            updateResult = courseDAO.update(updatedCourse);
            System.out.println(updateResult);
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }

    }
}
```

Section 8

## Transaction Management in JDBC

# Transaction

- **Transactions and ACID:**
  - ✓ A transaction is a single unit of work
- ACID properties describe transaction management:
  - ✓ **Atomicity** - all or nothing
  - ✓ **Consistency** - moves database between consistent states
  - ✓ **Isolation** - isolated from other transactions
  - ✓ **Durability** - once a transaction has been committed, it will remain so, even in the event of errors, power loss etc



## Transaction

- **Connection interface** provides methods to manage transaction.
- Transaction methods:
  - ✓ **setAutoCommit**(boolean) - true by default, each statement commits
  - ✓ **commit**() - Commits transaction
  - ✓ **rollback**() - Cancels transaction
- Advantages:
  - ✓ Fast performance - database hit at commit time

# JDBC Without Parameter

```
public void addCourse(Course course) {
    String sql = "INSERT INTO Course (course_id, subject_id, course_code, course_title, num_of_credits) " +
                 "VALUES ('" + course.getCourseId() + "', '" + course.getSubjectId() + "', '" +
        course.getCourseCode() + "', '" + course.getCourseTitle() + "', " + course.getNumOfCredits() + ")";
    try (
        Statement statement = connection.createStatement();) {
        statement.executeUpdate(sql);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

```
public class CourseAdder {
    public static void main(String[] args) {
        CourseDAO courseDAO = new CourseDAO();

        // Create a Course object with the course details to be added
        Course newCourse = new Course();
        newCourse.setCourseId("00005");
        newCourse.setSubjectId("002");
        newCourse.setCourseCode("CS102");
        newCourse.setCourseTitle("New Course Title");
        newCourse.setNumOfCredits(3);

        // Call the addCourse method to add the new course
        courseDAO.addCourse(newCourse);

        System.out.println("Course added successfully.");
    }
}
```

## setAutoCommit Example

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

// Insert the first course
String insertSql1 = "INSERT INTO Course (course_id, subject_id, course_code, course_title, num_of_credits) " +
                "VALUES ('00005', '005', 'CS105', 'Course 5', 3)";
statement.executeUpdate(insertSql1);

// Insert the second course
String insertSql2 = "INSERT INTO Course (course_id, subject_id, course_code, course_title, num_of_credits) " +
                "VALUES ('00006', '006', 'CS106', 'Course 6', 4)";
statement.executeUpdate(insertSql2);

// You can add more insert statements as needed

connection.commit();
connection.setAutoCommit(true);
```

# Transaction Example

```
public void insertCourses(List<Course> courses) {
    String insertSql = "INSERT INTO Course (course_id, subject_id, course_code, course_title, num_of_credits) "
                    + "VALUES (?, ?, ?, ?, ?";
    try {
        connection.setAutoCommit(false);
        PreparedStatement preparedStatement = connection.prepareStatement(insertSql);

        for (Course course : courses) {
            preparedStatement.setString(1, course.getCourseId()); preparedStatement.setString(2, course.getSubjectId());
            preparedStatement.setString(3, course.getCourseCode()); preparedStatement.setString(4, course.getCourseTitle());
            preparedStatement.setInt(5, course.getNumOfCredits());
            preparedStatement.executeUpdate();
        }
        connection.commit();
        connection.setAutoCommit(true);

        System.out.println("Inserted " + courses.size() + " courses successfully.");
    } catch (SQLException e) {
        try {
            connection.rollback();

            System.out.println("Rollback performed.");
        } catch (SQLException rollbackException) {
            rollbackException.printStackTrace();
        }
        e.printStackTrace();
    }
}
```

# Transaction Example

```
public class CourseInsertTest {
    public static void main(String[] args) {
        CourseDAO courseDAO = new CourseDAO();

        // Create a list of Course objects to insert
        List<Course> courses = new ArrayList<>();
        courses.add(new Course("00007", "007", "CS107", "Course 7", 3));

        courses.add(new Course("00008", "22007", "CS108", "Course 8", 4));

        // Call the insertCourses method to insert the courses
        courseDAO.insertCourses(courses);

    }
```

*subject_id exceeds 4 characters*

**Output:**

```
Rollback performed
```

Section 9

## Batch Processing in JDBC

09e-BM/DT/FSOFT - ©FPT SOFTWARE - FPT Software Academy - Internal Use

## Batch Processing

- Executes groups of queries together ==> improves performance
- **Statement** and **PreparedStatement** support batches
- Advantage: faster performance
- Batch Processing Methods:
  - ✓ **addBatch**(String) - Adds query to batch
  - ✓ **executeBatch**() - Executes batch of queries

## JDBC Batch with String Query

- **Step 1:**

```
connect.setAutoCommit(false);
```

- **Step 2:**

```
Statement statement = connect.createStatement();
statement.addBatch(<Insert query>);
statement.addBatch(<Insert query>);
statement.addBatch(<Update query>);
statement.addBatch(<Delete query>);
```

- **Step 3:**

```
int[] updateCounts =   statement.executeBatch();
connect.commit();
statement.close();
connect.setAutoCommit(true);
```

# JDBC Batch with PrepareStatement

```
public void insertCoursesInBatch(List<Course> courses) {

    String insertSql = "INSERT INTO Course (course_id, subject_id, course_code, course_title, num_of_credits)  VALUES (?, ?, ?, ?, ?)";
    try {
        connection.setAutoCommit(false); // Disable auto-commit
        PreparedStatement preparedStatement = connection.prepareStatement(insertSql);
        for (Course course : courses) {
            preparedStatement.setString(1, course.getCourseId());
            preparedStatement.setString(2, course.getSubjectId());
            preparedStatement.setString(3, course.getCourseCode());
            preparedStatement.setString(4, course.getCourseTitle());
            preparedStatement.setInt(5, course.getNumOfCredits());

            preparedStatement.addBatch();
        }
        int[] updateCounts = preparedStatement.executeBatch();
        connection.commit(); // Commit the transaction
        System.out.println("Inserted " + updateCounts.length + " courses successfully.");

        connection.setAutoCommit(true); // Enable auto-commit
    } catch (SQLException e) {
        try {
            connection.rollback(); System.out.println("Rollback performed.");
        } catch (SQLException rollbackException) {
            rollbackException.printStackTrace();
        }
        e.printStackTrace();
    }
}
```

# JDBC Batch with PrepareStatement

```
public class CourseBatchInsertTest {
    public static void main(String[] args) {

        CourseDAO courseDAO = new CourseDAO();

        // Create a list of Course objects to insert
        List<Course> courses = new ArrayList<>();
        courses.add(new Course("00009", "009", "CS109", "Course 9", 3));
        courses.add(new Course("00010", "010", "CS110", "Course 10", 4));

        // Call the insertCoursesInBatch method to insert the courses
        courseDAO.insertCoursesInBatch(courses);

    }
}
```

## Lesson Summary

- DriverManager handles driver registration
- Statements execute SQL queries
- ResultSets hold query results
- PreparedStatements use parameterized SQL
- CallableStatements call procedures
- Transactions for ACID properties
- Batch processing for performance

## References

- *https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html*
- *https://www.geeksforgeeks.org/jdbc-tutorial/*
- *https://www.baeldung.com/java-jdbc*
- *https://www.javatpoint.com/java-jdbc*

# THANK YOU!