Database Integration and New Features in Java 21

Database Connectivity with JDBC

JDBC (Java Database Connectivity) is an API that enables Java applications to interact with relational databases. It provides a standard interface for connecting to and interacting with databases using SQL. Here's an introduction to JDBC, its architecture, and how to connect to a MySQL database, execute queries, and handle results.

Introduction to JDBC and its architecture

JDBC Architecture

JDBC operates on a layered architecture involving several key components:

1. JDBC API: Provides the application-level interface for interacting with databases. It includes classes and interfaces like DriverManager, Connection, Statement, PreparedStatement, ResultSet, etc.
2. JDBC Driver: Acts as a bridge between the JDBC API and the database. It translates JDBC calls into database-specific calls. There are different types of JDBC drivers:
    a. Type 1: JDBC-ODBC Bridge Driver (deprecated in Java 8).
    b. Type 2: Native-API Driver (partially Java-based, relies on native code).
    c. Type 3: Network Protocol Driver (uses a middleware server).
    d. Type 4: Thin Driver (pure Java driver, converts JDBC calls directly to database protocol).
3. Database: The relational database system (like MySQL) that stores data and responds to SQL queries.

JDBC Workflow

1. Load the JDBC Driver: Register the JDBC driver class.
2. Establish a Connection: Use DriverManager to connect to the database.
3. Create a Statement: Use Connection to create Statement or PreparedStatement.
4. Execute Queries: Use Statement to execute SQL queries.
5. Process Results: Retrieve and process results from ResultSet.
6. Close Resources: Close the ResultSet, Statement, and Connection to release resources.

Connecting to MySQL database, executing queries, and handling results

1. Setting Up MySQL and JDBC Driver
    a. Download and Install MySQL: Ensure MySQL server is installed and running.
    b. Add JDBC Driver: Include the MySQL JDBC driver (mysql-connector-java-x.x.x.jar) in your project's classpath.
2. Sample JDBC Code

Connecting to MySQL Database

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

public class JDBCConnectionExample {
    public static void main(String[] args) {
        String url =
"jdbc:mysql://localhost:3306/your_database";
        String user = "your_username";
        String password = "your_password";

        try {
            // Load and register MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Establish a connection
            Connection connection =
DriverManager.getConnection(url, user, password);
            System.out.println("Connected to the database!");

            // Close the connection
            connection.close();
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Executing Queries and Handling Results

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

public class JDBCQueryExample {
    public static void main(String[] args) {
        String url =
"jdbc:mysql://localhost:3306/your_database";
        String user = "your_username";
        String password = "your_password";

        try {
            // Load and register MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Establish a connection
            Connection connection =
DriverManager.getConnection(url, user, password);

            // Create a Statement
            Statement statement =
connection.createStatement();

            // Execute a query
            String query = "SELECT id, name, email FROM
users";
            ResultSet resultSet =
statement.executeQuery(query);

            // Process the results
            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);
            }

            // Close resources
            resultSet.close();
            statement.close();
            connection.close();
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

Using PreparedStatement for Parameterized Queries

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

public class JDBCPreparedStatementExample {
    public static void main(String[] args) {
        String url =
"jdbc:mysql://localhost:3306/your_database";
        String user = "your_username";
        String password = "your_password";

        try {
            // Load and register MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Establish a connection
            Connection connection =
DriverManager.getConnection(url, user, password);

            // Create a PreparedStatement
            String query = "SELECT id, name, email FROM users
WHERE id = ?";
            PreparedStatement preparedStatement =
connection.prepareStatement(query);
            preparedStatement.setInt(1, 1); // Set the
parameter value

            // Execute the query
            ResultSet resultSet =
preparedStatement.executeQuery();

            // Process the results
            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);
            }
```

```
            // Close resources
            resultSet.close();
            preparedStatement.close();
            connection.close();
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }
}
```

New Features in Java 21

Java 21, released in September 2023, includes several new features and enhancements that improve performance, simplify coding, and provide new capabilities. Here's an overview of the latest features and their practical applications.

Overview of the latest features in Java 21

1. **Record Patterns**:

   Description: Introduces record patterns, allowing for deconstructing records in a more concise and readable way. This enhancement builds on the record types introduced in Java 14 and enhances their usability in pattern matching.

```
record Point(int x, int y) {}

public class RecordPatternExample {
    public static void main(String[] args) {
        Point point = new Point(1, 2);

        if (point instanceof Point(int x, int y)) {
            System.out.println("x: " + x + ", y: " + y);
        }
    }
}
```

2. **Pattern Matching for Switch**:

   Description: Extends pattern matching capabilities to switch expressions and statements, making code more concise and readable when dealing with complex conditional logic.

```
public class SwitchPatternExample {
    public static void main(String[] args) {
        Object obj = "hello";

        String result = switch (obj) {
            case Integer i -> "Integer: " + i;
            case String s -> "String: " + s;
            case null -> "null";
            default -> "Unknown";
        };

        System.out.println(result);
    }
}
```

3. **Virtual Threads (Project Loom)**:

   Description: Introduces virtual threads as a new concurrency model, enabling high-throughput, scalable applications with less overhead compared to traditional threads.

```
import java.util.concurrent.Executors;

public class VirtualThreadsExample {
    public static void main(String[] args) throws
InterruptedException {
        try (var executor =
Executors.newVirtualThreadPerTaskExecutor()) {
            for (int i = 0; i < 10; i++) {
                executor.submit(() -> {
                    System.out.println("Virtual thread: " +
Thread.currentThread());
                });
            }
        }
    }
}
```

4. **Structured Concurrency (Project Loom)**:

   Description: Simplifies handling of multiple concurrent tasks by providing a unified way to manage their lifecycle and error handling. This feature works with virtual threads to improve concurrency management.

```
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class StructuredConcurrencyExample {
```

```
    public static void main(String[] args) throws Exception {
        try (var scope =
Executors.newVirtualThreadPerTaskExecutor()) {
            Future<String> future = scope.submit(() -> {
                return "Hello, world!";
            });
            System.out.println(future.get());
        }
    }
}
```

5. **Enhanced Switch Expressions**:

   Description: Adds new features to switch expressions, including enhanced support for handling null values and new case labels.

```
public class EnhancedSwitchExample {
    public static void main(String[] args) {
        String day = "MONDAY";

        String activity = switch (day) {
            case "MONDAY" -> "Start of the week";
            case "FRIDAY" -> "End of the week";
            case "SATURDAY", "SUNDAY" -> "Weekend";
            default -> "Regular day";
        };

        System.out.println(activity);
    }
}
```

6. **Sealed Interfaces and Classes Enhancements**:

   Description: Expands the sealed classes and interfaces feature, allowing more flexibility in defining permitted subclasses and implementing more controlled hierarchies.

```
public sealed interface Shape permits Circle, Rectangle {}

public final class Circle implements Shape {
    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }
}

public final class Rectangle implements Shape {
```

```
    private final double width;
    private final double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
}
```

Practical applications of new features

1. **Record Patterns**:

   Application: Simplifies code that involves extracting values from records. Useful in scenarios where records are used to represent immutable data and pattern matching is required.

2. **Pattern Matching for Switch**:

   Application: Reduces boilerplate code in switch statements, making code easier to read and maintain. Ideal for scenarios where complex conditional logic needs to be handled.

3. **Virtual Threads (Project Loom)**:

   Application: Enables handling large numbers of concurrent tasks with minimal overhead. Useful in high-throughput applications such as web servers, real-time data processing, and concurrent applications.

4. **Structured Concurrency (Project Loom)**:

   Application: Simplifies the management of concurrent tasks and improves error handling. Suitable for applications that require coordination of multiple concurrent operations, like batch processing and parallel computation.

5. **Enhanced Switch Expressions**:

   Application: Provides a more flexible and concise way to handle different cases in switch expressions. Useful for scenarios requiring a comprehensive handling of various cases with reduced verbosity.

6. **Sealed Interfaces and Classes Enhancements**:

   Application: Enforces more controlled hierarchies and improves type safety. Ideal for defining domain models where certain classes or interfaces should be restricted to a specific set of implementations.