

# Module 2 Day 7



Java



spring

The DAO Pattern

# Module 2 Day 7 Lecture 6

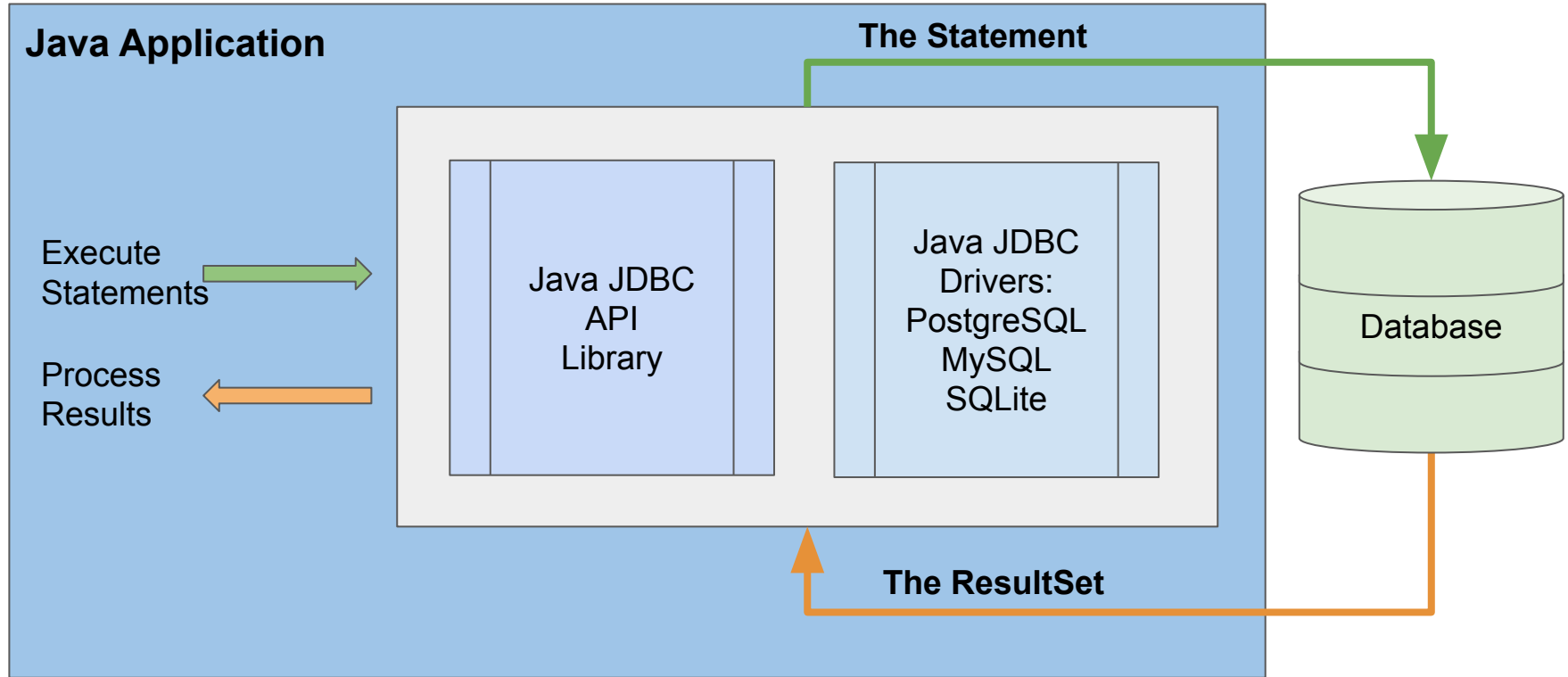
## Can you ... ?

- ... design a basic Database in 3NF
- ... create a database connection in Java
- ... execute SQL statements using that connection
- ... create and run Parameterized Queries
- ... explain the DAO Pattern

# JDBC Basics:

## A Traditional Approach

# JDBC: Java Database Connectivity



# JDBC Introduction

JDBC (Java Database Connectivity) is an API that is part of Java; it is used to create connections to a database from code.

- The goal today is to become familiar with and understand the collaborator classes and methods that are needed to talk to a Postgresql database.
- This pattern forms the basis for all applications that rely on persistent storage of data, including: Client Server, Classic JSP, and MVC.

# JDBC: The DataSource Class

- The DataSource class is responsible for creating the database connection
- DataSource has 4 commonly used methods:
  - **.setURL(<<String with URL>>)**: Sets the network location of the database, this can be any url, including your own workstation. The standard Postgres URL on our machines is: *'jdbc:postgresql://localhost:5432/<database>'*
  - **.setUsername(<<Username String>>)**: Sets the username for the database. *'postgres'*
  - **.setPassword(<<Password String>>)**: Sets the password for the database. *'postgres1'*
  - **.getConnection()**: returns a connection object that will be used for running queries.

```
BasicDataSource dataSource = new BasicDataSource();  
dataSource.setUrl("jdbc:postgresql://localhost:5432/dvdstore");  
dataSource.setUsername("postgres");  
dataSource.setPassword("postgres1");
```

Note: There is a more elegant way to manage the credentials and connection strings. This will be a topic for when we discuss “Dependency Injection” in module 3.

# JDBC: The Connection Class

- The Connection class creates a session for any database transactions.
- Connections are “pooled,” meaning that they are stored within the JVM’s memory and reused during a given session. This reduces the amount of “heavy lifting” processing needed to establish a database connection.
- The Connection object is instantiated using the getConnection() method of the DataSource object:

```
Connection conn = dataSource.getConnection();
```

# JDBC: The Statement Class

- The Statement class is responsible for the execution of the actual SQL commands.
- The Statement object can be instantiated by calling `createStatement()` method of the connection object

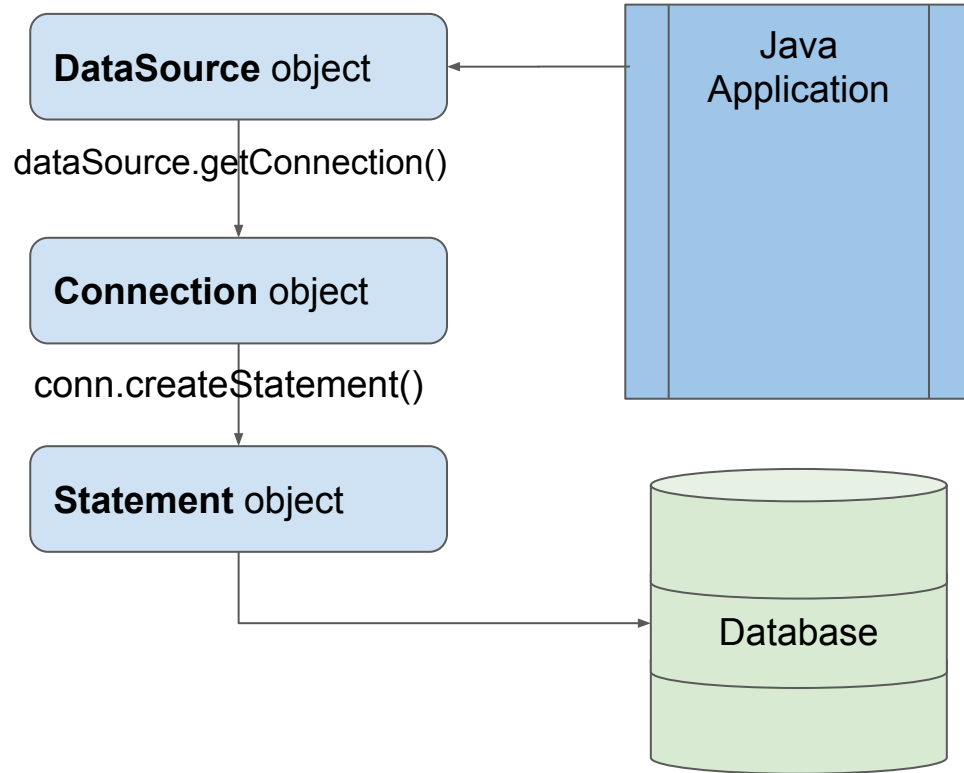
```
Statement stmt = conn.createStatement();
```

- The statement object's `.executeQuery(<<SQL Statement String>>)` method is used to run the SQL command.

```
String aSQLStatement = "SELECT name from country";  
stmt.executeQuery(aSQLStatement);
```



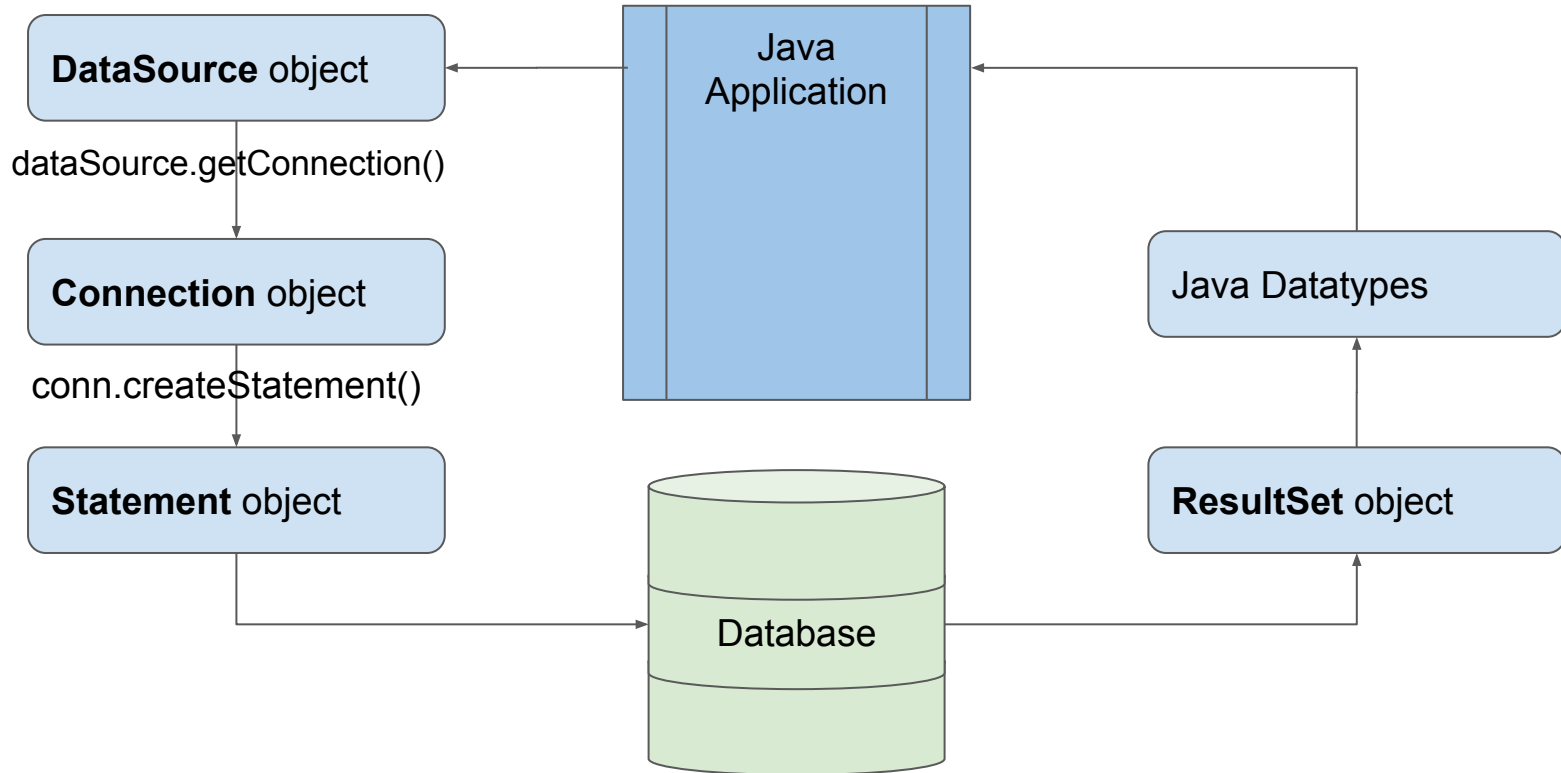
# The JDBC Flow in Java so far...



# The ResultSet Class

- The ResultSet class is the collaborator in charge of storing the query results from the Database.
- It contains several meaningful methods:
  - **.next()**: This method allows for iteration if the SQL operation returns multiple rows. Using next is very similar to the way we dealt with file processing (Scanner.nextLine).
  - **.getString(<<name of column in SQL result>>)** ,  
**getInt(<<name of column in SQL result>>)**,  
**getBoolean(<<name of column in SQL result>>)** ,etc. :  
These all get the values for a given column, strongly typed, for a given row.

# The JDBC Flow in Java



A group of children are walking along a path in a park, passing a row of Toy Soldiers. The soldiers are dressed in green uniforms and helmets, standing on green bases. The children are dressed in casual clothing like t-shirts and shorts. In the background, there are colorful playground structures, including a yellow and green 'Play Family' station and a red roller coaster track. The scene is bright and sunny.

Let's Code Classic JDBC

# Spring JDBC

# Spring JDBC Introduction

The traditional JDBC approach requires multiple steps and collaborators, a process that is repetitive and could be error prone. The Spring JDBC pattern simplifies the process.

- Spring is a popular Java framework that abstracts various operations (i.e. querying a database) to a higher level such that it's easier for developers to work with.
- Spring provides a **JdbcTemplate** class that accomplishes the previous operation in less lines of code.

# JdbcTemplate Class

- The JDBC template's constructor requires a data source. You can pass it the same data source object described in the regular JDBC workflow:

```
BasicDataSource dataSource = new BasicDataSource();  
dataSource.setUrl("jdbc:postgresql://localhost:5432/dvdstore");  
dataSource.setUsername("postgres");  
dataSource.setPassword("postgres1");  
  
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource)
```

# JdbcTemplate Class

- The jdbcTemplate class encapsulates connection and datasource management.
- The .queryForRowSet(<<String containing SQL>>)method will execute the SQL query.
  - Extra parameter constructor are available as well, allowing for any prepared statement placeholders.

```
String sqlString = "SELECT name from country";  
SqlRowSet results = jdbcTemplate.queryForRowSet(sqlString);
```

- For UPDATE, INSERT, and DELETE statements we will use the **.update** method instead of the .queryForRowSet method.

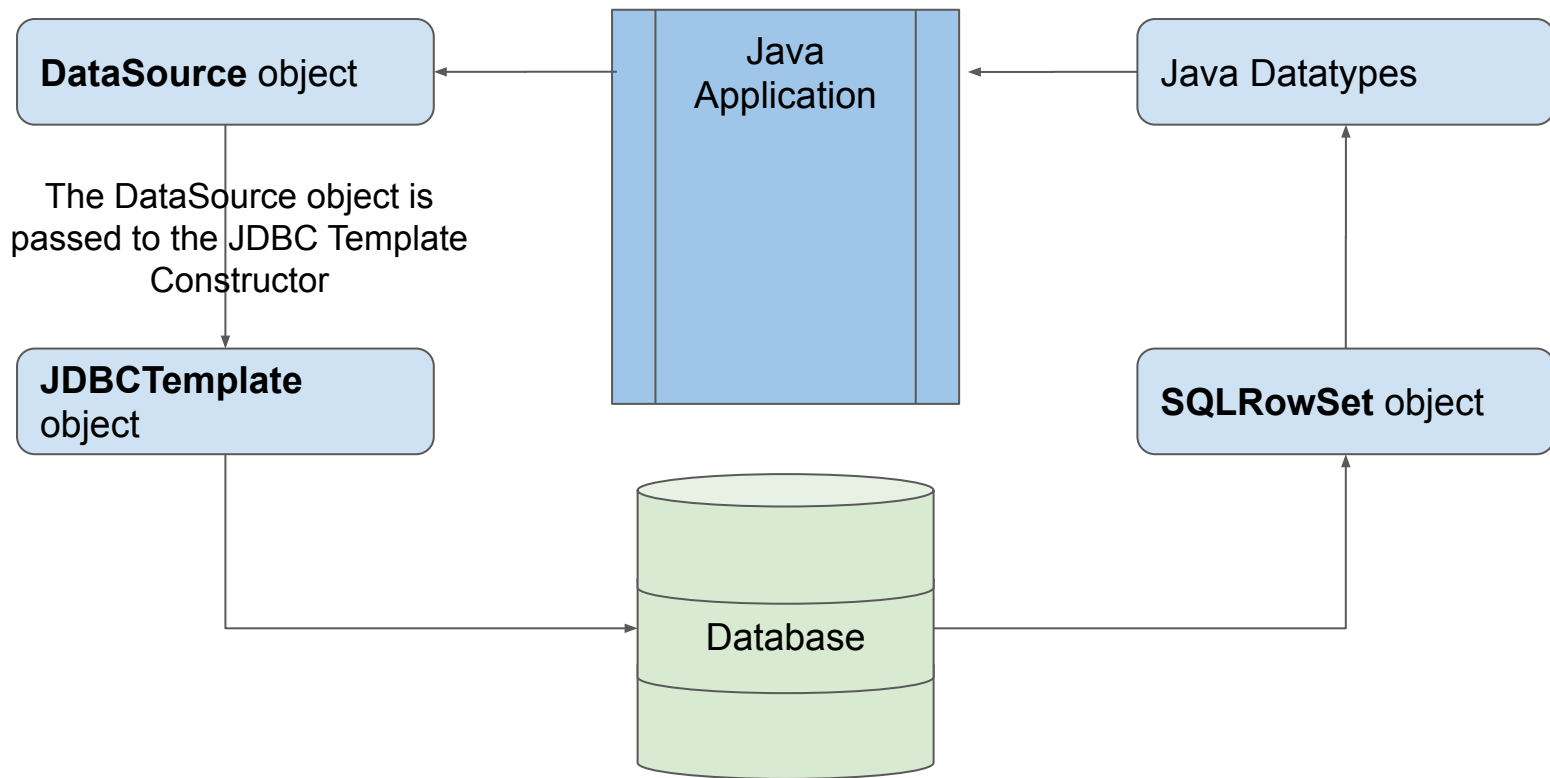
```
SqlRowSet results = jdbcTemplate.update(sqlString);  
// Used when the sqlString contains an UPDATE, INSERT, or  
DELETE.
```



# JdbcTemplate Class

- The results are stored in an object of class `SqlRowSet`, this behaves the exact same way as a `ResultSet` object:
- The same methods available to `ResultSet` are also available here:
  - **.next()**: This method allows for iteration if the SQL operation returns multiple rows. Using `next` is very similar to the way we dealt with file processing.
  - **.getString(<<name of column in SQL result>>)** , **getInt(<<name of column in SQL result>>)**, **getBoolean(<<name of column in SQL result>>)** ,etc. : These get the values for a given column, for a given row.

# The Spring JDBCTemplate Flow



A photograph of three Toy Soldiers standing in a line in a theme park. They are wearing their signature red uniforms and helmets. The soldier on the left is saluting, the middle one is also saluting, and the one on the right is in a dynamic pose. Behind them is a large, colorful structure resembling a rocket or a space land attraction, with a red and yellow striped section. The sky is blue with white clouds. The text "Let's advance to Spring JDBC" is overlaid in the center of the image.

Let's advance to Spring JDBC

# ORM and the DAO Pattern:

Using SpringJDBC to Populate, Interrogate, and  
Manipulate Application Objects

# DAO Pattern

- A database table can sometimes map fully or partially to an existing class in Java. This is known as ORM: **Object-Relational Mapping**.
- Object Relational Mapping is accomplished with a design pattern called DAO, which is short for **Data Access Object**.
- The DAO pattern also allows for loosely coupled systems by using Interfaces for database operations; future changes to our data infrastructure (i.e. migrating from 1 database platform to another) have minimal changes on the our business logic.

# DAO Pattern Step 1

- Start with an Interface that will establish the contract for database communications. The method signatures will typically perform basic CRUD operations for that Object type (i.e. search, update, delete).

```
public interface CityDAO {  
    public void save(City newCity);  
    public City findCityById(long id);  
}
```

Next, we will need to go ahead and create a concrete class that implements the DAO interface:

# DAO Pattern Step 2

```
public class JDBCCityDAO implements CityDAO {  
  
    private JdbcTemplate jdbcTemplate;  
  
    public JDBCCityDAO(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
  
    @Override  
    public void save(City newCity) {  
        String sqlInsertCity = "INSERT INTO city(id, name, countrycode, district, population) " +  
                                "VALUES(?, ?, ?, ?, ?)";  
  
        newCity.setId(getNextCityId());  
        jdbcTemplate.update(sqlInsertCity, newCity.getId(), newCity.getName(), newCity.getCountryCode(), newCity.getDistrict(), newCity.getPopulation());  
    }  
  
    @Override  
    public City findCityById(long id) {  
        City theCity = null;  
        String sqlFindCityById = "SELECT id, name, countrycode, district, population " +  
                                  "FROM city " +  
                                  "WHERE id = ?";  
  
        SqlRowSet results = jdbcTemplate.queryForRowSet(sqlFindCityById, id);  
        if(results.next()) {  
            theCity = mapRowToCity(results);  
        }  
        return theCity;  
    }  
}
```

The contractual obligations of the interface are met.

# DAO Pattern Step 3

- In our orchestrator class, we will be using a polymorphism pattern to declare our DAO objects:

```
CityDAO dao = new JDBCCityDAO(worldDataSource);
```

The Interface Reference

The Concrete Class Constructor



# DAO Pattern Step 4


- In our orchestrator class, we will be using a polymorphism pattern to declare our DAO objects:

```
City smallville = new City();  
smallville.setCountryCode("USA");  
smallville.setDistrict("KS");  
smallville.setName("Smallville");  
smallville.setPopulation(42080);
```

```
dao.save(smallville);
```

```
City theCity = dao.findCityById(smallville.getId());
```

We can now call the methods that are defined in concrete class and required by the interface.





Finally, we found the DAO ...