Module 2-6

JDBC and DAO Pattern

Spring JDBC

JDBC Introduction

- JDBC stands for Java Database Connectivity, and it's a series of specifications for allowing a Java program to interact with a database via a driver.
- Spring is a popular Java framework that implements (amongst other things) JDBC.
- In summary:
  - We use Spring JDBC, which is an implementation of JDBC, which contains JDBC Drivers, which connect to the database.

The BasicDataSource class

- The BasicDataSource class defines the database’s location and credentials.

```
BasicDataSource dataSource = new BasicDataSource();

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

- Here we created an instance of the BasicDataSource class, and used its setters to provide the database location, username, and password.

JDBCTemplate Class (Instantiating)

The JDBCTemplate class provides the means by which a query can be made to the database and the results retrieved.

- The constructor for the JDBC Template requires that we pass in a data source object (which we talked about in the previous slide)

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource)
```

JDBCTemplate Class (Sending a SQL Query)

- 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.

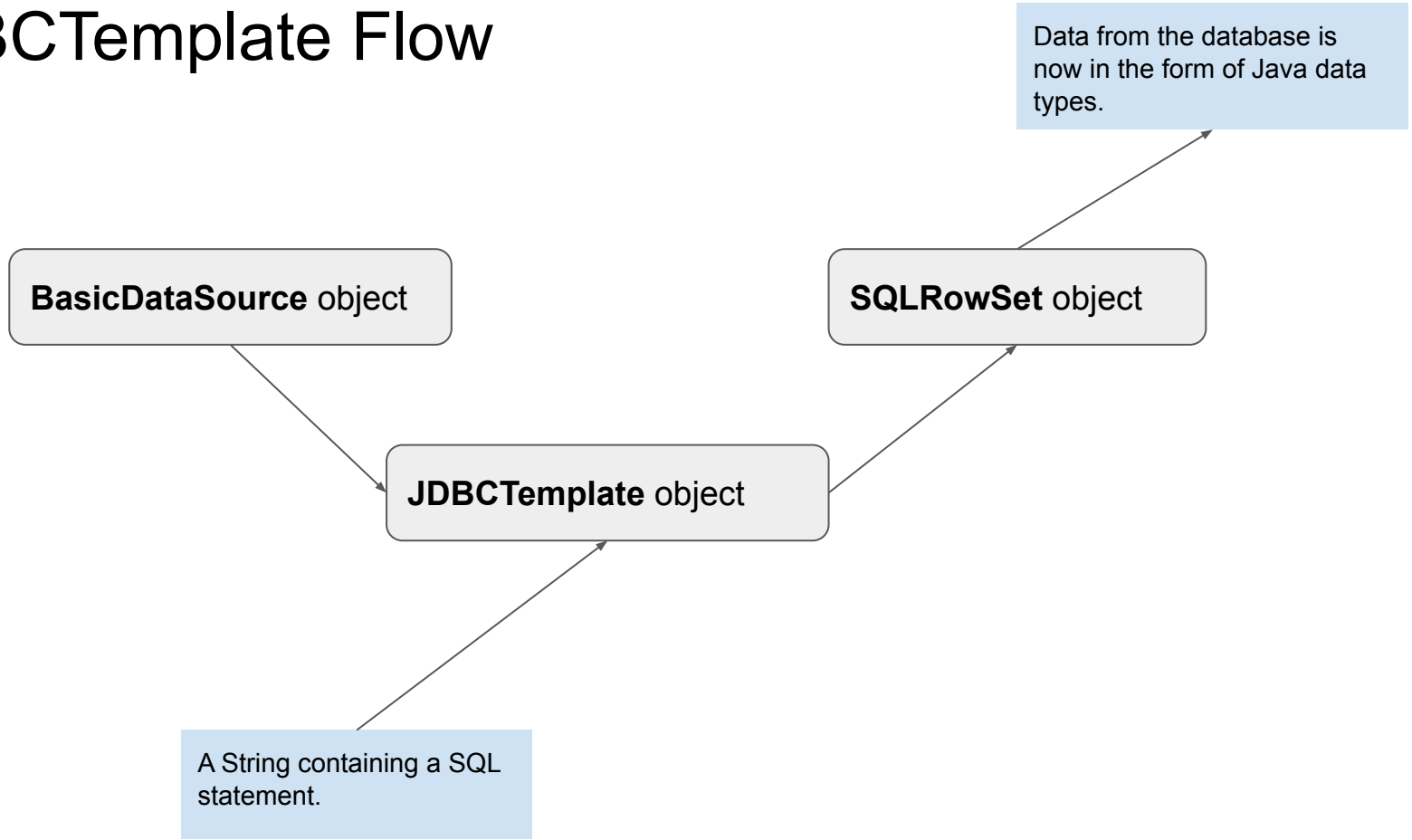
```
SqlRowSet results = jdbcTemplate.update(sqlString);
// Where sqlString contains an UPDATE, INSERT, or DELETE.
```

SQLRowSet Class (Accessing the Results)

The RowSET class has the following methods:

- **next()**: This methods 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.

JDBCTemplate Flow



Let’s do a quick example.

DAO Pattern

DAO Pattern

- A database table can sometimes map fully or partially to an existing class in Java. This is known as **Object-Relational Mapping**.

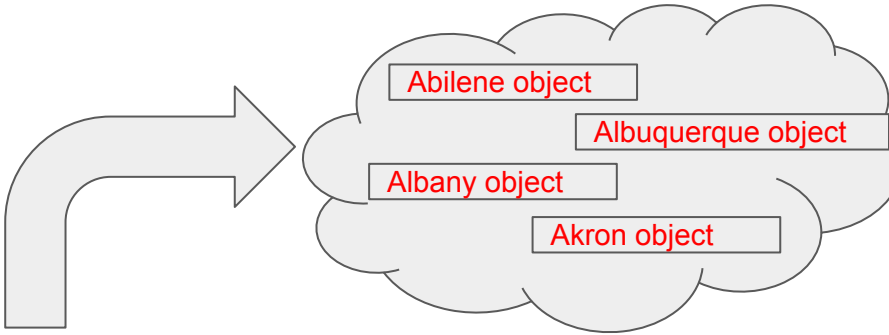
- We implement Object Relation Mapping with a design pattern called DAO, which is short for **Data Access Object**.

- We do this in a very specific way using Interfaces so that future changes to our data infrastructure (i.e. migrating from 1 database platform to another) are easier to manage.

DAO Pattern (Setup)

First, we have a class in Java that corresponds to the columns being retrieved from the database:

```
public class City {
        private long cityId;
        private String cityName;
        private String stateAbbreviation;
        private long population;
        private double area;
        // + getters & setters
}
```



*	city_id	city_name	state_abbreviation	population	area
1	1	Abilene	TX	123420	276.4
2	2	Akron	OH	197597	160.6
3	3	Albany	NY	96460	56.8
4	4	Albuquerque	NM	560513	487.4
5	5	Alexandria	VA	159428	38.8
6	6	Allen	TX	105623	70.2
7	7	Allentown	PA	121442	45.3
8	8	Amarillo	TX	199371	262.6
9	9	Anaheim	CA	350365	129.5

Each row of data becomes an object, an instance of City.

The instance variables of our class match the columns of our query

DAO Pattern Step 1

- We start off with an Interface specifying that a class that chooses to implement the interface must implement methods to communicate with a database (i.e. search, update, delete). Consider the following example:

```
public interface CityDao {

    City getCity(long cityId);
    void createCity(City city);
}
```

DAO Pattern Step 2

- Next, we want to go ahead and create a concrete class that implements the interface, this concrete class (let's say it's called JdbcCityDao) needs to implement the following 2 methods:

```
@Override
public City getCity(long cityId) {
    City city = null;
    String sql = "SELECT city_id, city_name, state_abbreviation, population, area " +
                 "FROM city " +
                 "WHERE city_id = ?";
    SqlRowSet results = jdbcTemplate.queryForRowSet(sql, cityId);
    if (results.next()) {
        city = mapRowToCity(results);
    }
    return city;
}
```

Note that the sql String has a placeholder value, denoted by the question mark. This hole is plugged by the jdbcTemplate using the variable cityId.

```
@Override
public void createCity(City city) {
    String sql = "INSERT INTO city (city_name, state_abbreviation, population, area) " +
                 "VALUES (?, ?, ?, ?)";
    Long newId = jdbcTemplate.update(sql,
     city.getCityName(), city.getStateAbbreviation(), city.getPopulation(), city.getArea());

}
```

Note how the various placeholders have their values substituted by the city object's getters.

Creating an INSERT with a return

You can have a JDBC method return an object of a particular type upon completion of the query. Here we have a long, representing the city ID being returned after the INSERT.

```
@Override
 public City createCity(City city) {
     String sql = "INSERT INTO city (city_name, state_abbreviation, population, area) " +
                  "VALUES (?, ?, ?, ?) RETURNING city_id;";
     Long newId = jdbcTemplate.queryForObject(sql, Long.class,
           city.getCityName(), city.getStateAbbreviation(), city.getPopulation(), city.getArea());

     return getCity(newId);
 }
```

Let’s implement a DAO class

DAO Pattern Step 3

- In our driver class, we will be using polymorphism to declare our DAO objects:

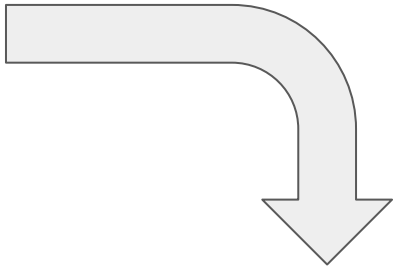
```
CityDAO dao = new JDBCCityDAO(dataSource);
```

DAO Pattern Step 3

- We can now call the DAO methods we declared to interact with the database:

Driver class

```
cityDao.createCity(newCity);
```



In this case we call the DAO's createCity method, while providing an argument. The argument newCity is an object of type city

DAO class

```
@Override
 public void createCity(City city) {
    String sql = "INSERT INTO city (city_name, state_abbreviation, population, area) " +
            "VALUES (?, ?, ?, ?)";
    Long newId = jdbcTemplate.update(sql,
     city.getCityName(), city.getStateAbbreviation(), city.getPopulation(), city.getArea());

 }
```

Let’s now use the DAO class